Effective Heuristic-based Test Generation Techniques for Concurrent Software

by

Niloofar Razavi

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Effective Heuristic-based Test Generation Techniques for Concurrent Software

Niloofar Razavi

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2014

With the increasing dependency on software systems, we require them to be reliable and correct. Software testing is the predominant approach in industry for finding software errors. There has been a great advance in testing sequential programs throughout the past decades. Several techniques have been introduced with the aim of automatically generating input values such that the executions of the program with those inputs provide meaningful coverage guarantees for the program.

Today, multi-threaded (concurrent) programs are becoming pervasive in the era of multi-processor systems. The behaviour of a concurrent program depends not only on the input values but also on the way the executions of threads are interleaved. Testing concurrent programs is notoriously hard because often there are exponentially large number of interleavings of executions of threads that has to be explored. In this thesis, we propose an array of heuristic-based testing techniques for concurrent programs to prioritize a subset of interleavings and test as many of them as possible. To that end, we develop:

(A) a sound and scalable technique that based on the events of an observed execution, predicts runs that might contain null-pointer dereferences. This technique explores the interleaving space (based on the observed execution) while keeping the input values fixed and can be adapted to predict other types of bugs.

(B) a test generation technique that uses a set of program executions as a program under-approximation to explore both input and interleaving spaces. This technique generates tests

that increase branch coverage in concurrent programs based their approximation models.

(C) a new heuristic, called bounded-interference, for input/interleaving exploration. It is defined based on the notion of data-flow between threads and is parameterized by the number of interferences among threads. Testing techniques that employ this heuristic are able to provide coverage guarantees for concurrent programs (modulo interference bound).

(D) a testing technique which adapts the sequential concolic testing to concurrent programs by incorporating the bounded-interference heuristic into it. The technique provides branch coverage guarantees for concurrent programs.

Based on the above techniques, we have developed tools and used them to successfully find bugs in several traditional concurrency benchmarks.

Acknowledgements

I want to take this opportunity to thank all the people that had an influence in the work presented in this thesis. First and foremost, none of this work would have been possible without Azadeh Farzan's support. I am grateful for the time she dedicated to my projects, as well as her inspiration. I would like to thank my committee members, Marsha Chechik, Sheila McIlraith, Steve Easterbrook, and Scott Stoller for their insightful comments and suggestions about my work.

I have to thank my collaborators Madhusudan Parthasarathy and Francesco Sorrentino from University of Illinois at Urbana Champaign for their collaboration on the null-pointer dereference prediction work, and Helmut Veith and Andreas Holzer from the Vienna University of Technology for their collaborations on (conc)^{2}olic testing work. I need to thank the people at the NEC Laboratories America, Aarti Gupta, Franjo Ivancic and Vineet Kahlon, for the amazing internship experience I had at NEC. I learned a lot from them and enjoyed every minute of working at NEC. A special thanks to all my Toronto friends and officemates: Varada Kolhatkar, Jocelyn Simmons, Golnaz Elahi, Alicia Grubb, Zachary Kincaid, Aws Albarghouthi and Michalis Famelis.

Finally and foremost, I would like to thank my family. My parents, Seyed Alireza and Sousan, whose constant love and encouragement has helped me to be optimistic even in the most difficult days of my life. My sisters, Negin and Negar, who were always listening to my complains patiently. My love, Andreas, whose unwavering love, support, and understanding made the experience easier.

Contents

List of Tables

List of Figures

Appendices

Chapter 1

Introduction

1.1 Background

Software systems nowadays affect every aspect of our life; they are used in medical services, transportation, education systems, businesses, and etc. Even a small error in any of these systems may lead to huge loss of money, time, or lives. Therefore, there is a great need to develop techniques to ensure that software systems are reliable, safe, and secure. In industry, software testing is still the predominant technique to find correctness and performance issues in software systems. Billions of dollars are spent on software testing each year which includes half of the cost of software development [20].

Sequential software systems consist of a single thread of execution. The testing process of a sequential system includes providing different input values to the system and investigating the behaviour of the system under the given inputs. For example, *Concolic testing* [24, 73, 5, 84, 4] is an automatic sequential testing technique which executes the program with both concrete and symbolic inputs simultaneously and uses path constrains (i.e., branch conditions encountered during execution) on symbolic inputs to generate input values that lead the execution towards uncovered parts of the program. Sequential software testing techniques are often coupled with a notion of *coverage* that the technique provides. Various coverage criteria have been introduced

for sequential program testing over the years, e.g., path coverage [24, 4, 73], control-flow coverage [84], predicate coverage [34], etc. These coverage criteria quantify the testing process and give the tester some meaningful information about how much the software has been tested.

Nowadays, multi-threaded (concurrent) software systems are becoming prevalent due to the increase in the usage of multi-core hardware. However, testing concurrent software is more challenging than testing sequential software since the behaviour of a concurrent program not only depends on input values but also is affected by the way the executions of threads are interleaved; i.e., even under fixed input values, different interleavings of executions of threads may lead to different behaviours. *Test generation* for concurrent systems includes both (i) exploring the input space to find a set of input values that *may* trigger a bug (*input generation*), and (ii) exploring the interleaving space, with bug-triggering input values, to find possible bugs (*schedule generation*). However, the exploration space is huge for real programs and it is infeasible to fully explore both input and interleaving spaces.

Stress testing [2] and *randomized* testing [11, 83, 71, 58, 43, 3] are two traditional testing techniques for concurrent programs. Stress testing [2] is an approach which puts programs under pressure by providing a set of input values to the program and then executing the program with each input for a long time (for server applications) or for many times with many threads (for other types of applications) with the hope of exploring different interleavings in different executions. Randomized testing [11, 83, 71, 58, 43, 3], on the other hand, aims at exploring distinct interleavings in different executions by changing the priorities of the threads on the fly or strewing the code with sleep commands for random time intervals. However, both of these techniques are highly ineffective in finding concurrency bugs.

More recent techniques employ some heuristics to restrict the exploration space to a manageable subset of input values or interleavings. *Heuristic-based* techniques can be categorized as follows:

(i) Interleaving exploration based on fixed inputs:

Techniques in this category rely on a given set of input values and focus on interleaving exploration while keeping the inputs fixed. For example, there is an array of *prediction* techniques [17, 82, 35, 93, 92, 90, 89, 7, 59, 74, 68] which are based on the philosophy of using heuristics to target interleavings that are more likely to contain bugs, e.g., interleavings that contain *data races* [13, 95, 57, 50, 69, 18] or *atomicity violations* [75, 58, 49, 94, 19, 15], and testing as many of those as possible (under the given time and space limitations).

A data race occurs when two threads access a shared memory location at the same time and at least one of the accesses is a write access. Data races might lead to unpredictable program behaviors and could be symptomatic of errors. A code unit is not atomic if it is interrupted during an execution by statements from another threads, and the interaction cannot be ruled out as harmless by presenting an equivalent execution in which this interruption does not occur. Atomicity violations could lead to concurrent program behaviours overlooked by programmers [47]. Many prediction techniques use data races and atomicity violations as heuristics to reduce the interleaving exploration space [82, 90, 59, 68]. Prediction techniques utilize a static lock-based [17, 82, 35, 59, 93, 92] or symbolic [90, 89] (using SMT solvers) analysis on a *single* observed run of the program to predict buggy runs.

There is also an array of search techniques [53, 54, 12, 55] that take a more coverage-oriented approach than the prediction techniques. They characterize a subset of the search space by a bounding parameter p. More behaviors are explored as p is increased, and in the limit all behaviors are explored. *Context bounding* [53, 54, 55] has been used as a heuristic to prioritize interleavings within a bounded number of context-switches over the others. The intuition behind this search strategy is that many bugs in concurrent programs manifest themselves by a few number of context-switches occurring during the program execution. For example, CHESS [53] is a tool that explores all interleavings (under the fixed input values) up to a bounded number of context-switches. *Delay bounding* [12] has been used to transform a deterministic scheduler into a sufficiently non-deterministic one (for the testing purpose) by

enabling it to delay each ready task for a bounded number of times. The non-deterministic scheduler allows efficient exploration of the interleaving space by increasing the possibility of exploring different program behaviours in different program executions.

The main advantage of the techniques in this category is that they are both simple and efficient in finding simple concurrency bugs that do not require complicated input values to be revealed. They simplify the testing process by ignoring input exploration and have been proven to be very effective in bug finding. As a result, these techniques are used for testing software in earlier development stages under some manually provided input values where the goal is to catch simple bugs as soon as possible.

(ii) Input/interleaving exploration based for *under-approximated programs*:

Techniques in this category use program *under-approximations* as heuristics to limit the exploration space; i.e., input/interleaving exploration is done for the approximated programs. Most of these techniques [80, 79] use *concurrent trace programs*, i.e., program slices built from program executions, as an under-approximation of concurrent programs. They perform a *static symbolic* analysis on concurrent trace programs, based on encoding possible executions of concurrent trace programs as a set of logical constraints and using SMT solvers, to check safety properties (of course they are incomplete for *proving* properties because of the approximations); e.g., the output of the analyses is whether an assertion can be violated or not.

The main advantage of these techniques over the techniques in the first category is that they perform input exploration (although it is limited to the approximated programs) and hence could be used to catch more complicated bugs; of course, this advantage comes at the expense of more complicated analyses. Therefore, these techniques are best to be applied after testing the software with the techniques in the first category to search for bugs that might be overlooked there, i.e., bugs that occur under specific input values that are not tested by the techniques in the first category.

(iii) Input/interleaving exploration for *programs*:

Techniques that fall into this category explore *both* input and interleaving spaces of the whole program while using some heuristics for interleaving exploration. *Sequentialization* techniques [44, 42, 85, 63, 62, 21] transform a concurrent program into a sequential program and then analyze the sequential program statically (e.g., for finding assertion violations) using sequential analysis techniques. These techniques utilize some heuristics to embed a subset of behaviours of the concurrent program in the resulting sequential program. Most of these sequentialization techniques are based on the context bounding heuristic; the sequential program encodes all program behaviours within a bounded number of context-switches by allowing a context-switch non-deterministically after each concurrent program statement if the bound has not been reached yet.

Other techniques in this category [73, 70] leverage concolic testing techniques, using some heuristics for interleaving exploration, to generate tests for concurrent programs. For example, jCute [73] is a concolic testing tool for concurrent Java programs that uses data races as a heuristic to prune the interleaving space. It executes the program and identifies data races in the observed execution. After each program execution, jCute either keeps the interleaving fixed as before and performs input generation to cover a previously uncovered part of the program or keeps the input values fixed as before and explores a new interleaving by simply re-ordering the events involved in a data race.

The techniques in this category are more expensive than the techniques in the second category since the input/interleaving exploration is performed for the whole program rather than its approximated model. However, by using an appropriate heuristic in interleaving exploration, these techniques are capable of providing coverage guarantees for the concurrent programs after the testing process is finished. Therefore, these techniques can be applied in later stages of software development (when techniques in other categories fail to find any more bugs) to provide quality assurance certifications for software.

1.2 Contributions

In this dissertation, we focus on effective test generation for concurrent programs and advance the state-of-the-art heuristic-based concurrent program testing techniques mentioned above. Specifically, we have the following contributions:

A. Sound and scalable prediction of null-pointer dereferences from a single program run

A prediction technique (in category (i)) is *sound* if the predicted runs are feasible program runs and is *scalable* if it works for large runs. Having an analysis which is both sound and scalable is a common challenge in all prediction techniques and often one of these issues is sacrificed for the benefit of the other. On the other hand, prediction techniques have mostly focused on data races, atomicity violations and assertion violations as heuristics to explore interleavings that might contain any of these violation patterns. However, the applicability of prediction techniques is not restricted to these bugs; i.e., they can target other types of bugs (e.g., memory bugs, deadlocks, and etc.) that are also common in concurrent programs. That requires to come up with appropriate violation patterns (reflecting these bugs) and provide corresponding analyses.

We introduce a new pattern, called *null reads*, for predicting *null-pointer dereferences* in concurrent programs. The intuition behind this pattern is that *null* is a critical value and in many cases reading null values might lead to memory bugs. Our prediction technique is both sound and scalable. To provide scalability, the analysis is performed at the shared communication level (i.e., accesses to shared variables and synchronization events) by suppressing local computation in the observed runs. We also develop a static pruning technique which drastically reduces the size of the prediction problem. To provide soundness, we employ the maximal causal model [74] which works at the shared communication level and guarantees soundness. We also develop a *relaxation* technique that allows us to deviate from the maxi-

mal causal model gradually to predict some (not necessarily sound) runs when the prediction problem has no answer in the maximal causal model.

We propose two different techniques for encoding the prediction problem based on the maximal causal model; in the first technique, the problem is encoded as a constraint satisfaction problem and the state-of-the-art SMT (Satisfiability Modulo Theories) solvers are used to search for solutions. The second technique is based on conceptualization and realization of the prediction problem as an AI automated planning [56] problem. This enables us to benefit from compact encoding techniques and advanced heuristic-based searching algorithms embedded in AI planners.

B. Test generation based on under-approximated programs

Most of the techniques that use program approximations to perform input/interleaving exploration (in category (ii)), so far, were aimed at finding assertion violations. In fact, none of these techniques have targeted test generation (i.e., input and schedule generation) for exploring different possible program behaviours. Furthermore, all existing techniques in category (ii) fix the approximation model a priori which does not allow exploring program code or behaviors that are beyond the approximation. For example, they cannot catch the violation of assertions that are not present in the approximation.

We use concurrent trace programs (i.e., program slices built from program executions) as program approximation models and develop a *multi-trace analysis* to generate tests for concurrent programs. The multi-trace analysis is built on top of symbolic prediction techniques and utilizes information available in multiple program runs to generate tests that would increase code coverage in concurrent programs.

However, we do not fix the approximation model a priori, i.e., the approximation is augmented by the observed run after running each generated test. Furthermore, we make the multi-trace analysis target test generation for branches that are not present in the approximation which allows us to explore program behaviours that are beyond the approximation. Note that in

an *active testing framework* [25], many runtime bugs can be encoded as branches. Therefore, by targeting branch coverage, one can implicitly aim for catching those bugs. We use this fact and combine a sequential testing technique with the multi-trace analysis such that individual threads are exposed to sequential test generation first to increase branch coverage as much as possible. Upon saturation, we fall back to our multi-trace analysis to generate tests for covering branches that are not covered in sequential testing.

C. Bounded-interference heuristic

Heuristics used by techniques in category (iii) have following problems:
(i) Inefficiency: most sequentialization techniques use the context bounding heuristic. Note that many thread interleavings might be equivalent to each other according to the way threads interfere with each other. Therefore, exploring all such interleavings reduces the efficiency without discovering any new bugs.
(ii) Lack of coverage guarantees: most concolic testing techniques for concurrent programs use data races as a heuristic for exploring the interleaving space. Due to this heuristic, these techniques are unable to quantify the partial work done during the testing process as a coverage measurement. Therefore, they cannot provide any coverage guarantees (on program code or behaviours) for concurrent programs when the time or memory limit is hit.

We introduce a new heuristic, called *bounded-interference*, for heuristic-based techniques in category (iii), to efficiently provide coverage guarantees for concurrent programs. An *interference* happens whenever a thread reads a value that is written by another thread. The idea behind the bounded-interference heuristic is to gradually explore all program behaviours within a bounded number of interferences among threads. This heuristic is parameterized with the number of interferences and therefore can be used to provide coverage guarantees (modulo the parameter bound) for concurrent programs.

Another property of this heuristic is that it is defined based on the notion of data flow among the threads, in contrast to the control-based notions such as context bounding that are

tied to schedules. Therefore, it can be naturally incorporated into sequential testing techniques to explore the input space and the interference space in a unified manner.

To verify the effectiveness of the bounded-interference heuristic in finding concurrency bugs, we develop a sequentialization technique which transforms a concurrent program into a sequential program such that the sequential program embeds all behaviours of the concurrent program within a bounded number of interferences. We keep the sequentialization technique simple by considering concurrent programs with only two threads where only one thread can be interfered by the other one.

A nice property of the sequentialization technique is that inputs of the concurrent program and interference scenarios are both encoded as inputs of the generated sequential program and hence the underlying sequential testing technique is able to explore both input and interference scenario spaces side by side. The sequentialization is sound, i.e., every bug in a generated sequential program represents a bug in the corresponding concurrent program and applying a sequential testing technique with specific coverage guarantees provides coverage guarantees (modulo the interference bound) on the concurrent program.

D. Concolic testing of concurrent programs with coverage guarantees

Existing concolic test techniques for concurrent programs (in category (iii)) use data races as a heuristic for exploring the interleaving space; i.e., interleaving exploration is done by switching the order of events involved in a data race in a previous execution. However, these techniques are able to provide coverage guarantees only when the testing algorithm is terminated after considering all possible orderings of events involved in a data race. Note that this exploration space is often very large for real world programs such that the testing algorithm fails to terminate in a reasonable amount of time. Unfortunately, due to the data race heuristic, these techniques are unable to quantify the partial work done (e.g., at the occasion of a timeout) as a meaningful coverage measure for the program.

We employ the bounded-interference heuristic in leveraging a sequential concolic testing

technique to generate tests for concurrent programs. Using the bounded-interference heuristic, our concolic testing technique provides coverage guarantees (modulo the interference bound) for concurrent programs both after the testing process is finished and when a time/computation limit is reached. We introduce a new component in concolic testing that explores possible interference scenarios (within the interference bound), and build a general framework which can employ different exploration strategies for inputs and interference scenarios.

We develop a search strategy that targets branch coverage in concurrent programs; i.e., interference scenario and input spaces are explored based on branches of the concurrent program that are yet uncovered during the testing process. This test generation technique is sound and provides branch coverage guarantees (modulo interference bound) for concurrent programs.

1.3 Outline

This thesis is organized as follows:

- Chapter 2 describes a sound and scalable technique for predicting null-pointer dereferences in concurrent programs. This chapter is based on two of our publications on this technique (i.e., [65] and [16]). We discuss the null reads pattern and present our static pruning which targets increasing scalability. Then, we present the logical constraints and planning encodings of the prediction problem based on the maximal causal model. We discuss the relaxation technique and experimentally show the effectiveness and efficiency of the prediction technique. Finally, we describe the related work and compare it to our null-pointer prediction technique.

- Chapter 3 is about the test generation technique based on under-approximated programs. This chapter is organized according to our publication on test generation based on multi-trace analysis (i.e., [66]). We present our multi-trace analysis in detail and show how it can be combined with a traditional sequential testing to efficiently increase branch coverage in concurrent programs. We experimentally evaluate the effectiveness of this

test generation technique in increasing branch coverage and finding concurrency. At the end of the chapter, we present the related work and compare it to our multi-trace analysis technique.

- Chapter 4 introduces the bounded-interference heuristic through some examples. We provide a theoretical comparison between the bounded-interference and context bounding heuristics. We show how this heuristic can be leveraged/employed by sequential testing techniques to generate tests with coverage guarantees for concurrent programs.

- Chapter 5 contains the sequentialization technique based on the bounded-interference heuristic. This chapter is based on our publication on bounded-interference sequentialization (i.e., [64]). We present the transformation algorithm and prove it to be sound and complete (depending on the coverage guarantees of the underlying sequential testing tool). We evaluate the effectiveness and efficiency of testing based on this sequentialization technique through a set of experiments. Finally, we provide an overview of the sequentialization techniques in the literature and compare them to our sequentialization technique.

- Chapter 6 describes our concolic testing technique for concurrent programs. This chapter is organized based on our publication on bounded-interference concolic testing of concurrent programs (i.e., [14]). We show how the general testing framework is built based on adapting a sequential concolic testing technique to concurrent programs by employing the bounded-interference heuristic in the search strategies. Then, we present algorithms for a search strategy that targets branch coverage in concurrent programs. We prove the algorithms to be sound and complete. We experimentally evaluate the effectiveness of the testing technique in code coverage and finding concurrency bugs. Finally, we present the related work and compare it to our bounded-interference concolic testing.

- Chapter 7 summarizes the research in thesis and identifies possible directions for future work.

Chapter 2

Predicting Null-Pointer Dereferences in Concurrent Programs

Prediction-based testing is a promising approach for testing concurrent programs [82, 59, 89, 90, 74, 68, 7, 77, 78]. It involves observing one execution of the program under test with some input values, and from that predict alternate interleavings of thread executions with the *same input values*. Prediction only explores the interleavings of thread executions that are close to the observed executions only, while at the same time explores interesting interleavings that are likely to lead to errors.

Prediction techniques, so far, have focused on predicting bugs that correspond to *atomicity violations* [82, 59, 90], *data races* [68], and *assertion violations* [89, 7]; i.e., these violation patterns are used as heuristics to reduce the exploration space to interleavings that are more probable to realize any of these patterns. Although these heuristics have been very successful in finding concurrency bugs, a recent research [96] shows that *memory* bugs (e.g., null-pointer dereferences) are often more harmful than many other types of bugs since they normally cause program crashes. Therefore, memory bugs could be good candidates to be targeted by prediction techniques.

Here, we propose a new violation pattern for prediction that is different from data races or

atomicity violations; we propose *null reads* that target interleavings that lead to null-pointer dereferences. Given an arbitrary execution of a concurrent program, we investigate fundamental techniques to *soundly* and *scalably* predict executions that are likely to realize null reads patterns:

1. **Approximation:** We use an approximation of the prediction problem that ignores local computation entirely to achieve scalability; we use the maximal causal model [74] which works at the shared communication level (i.e., accesses to shared variables and synchronization events). Our approximation of the prediction problem asks for runs that force threads to read *null* values where possible. Predicted runs in this model will be feasible but may not actually cause a null-pointer dereference (e.g., the thread reading a null value might dereference a pointer only if it is not null), though they are likely to do so.

2. **Static Pruning:** We use a static analysis that aggressively prunes the executions by identifying a small segment of the observed run on which the prediction effort can be focused. Pruning of executions does not affect feasibility of the runs, but increases the scalability of our technique.

3. **Relaxed prediction:** We utilize a formulation of the prediction at the shared communication level that allows some leeway so that the prediction algorithm can predict runs with mild *deviations* from the maximal causal model; this makes the class of predicted runs larger at the expense of possibly making them *infeasible*, though in practice, we found the majority of the predicted runs to be feasible.

4. **SMT and AI planning encodings:** We encode the prediction problem both as a constraint satisfaction problem and as an AI planning problem. The former enables the applicability of state-of-the-art SMT solvers while the latter enables us to benefit from the compact encoding and fast heuristic-based searching techniques available in AI planners by conceptualization and realization of the prediction problem as an AI automated planning [56] problem.

5. Re-execution: The runs predicted using the above techniques might be infeasible (in case of relaxed prediction), or might be feasible and yet not cause any null-pointer dereference. We mitigate this by re-executing the program according to the predicted runs to check if a null-pointer dereference actually occurs. Errors reported are always real (i.e., they cause an uncaught exception or result in failing the test harness), and hence we incur no false positives.

This chapter is based on our publications on these techniques (i.e., [65] and [16]). We elaborate these techniques in detail and evaluate them experimentally.

2.1 Motivating Example

Consider the code snippet extracted from the `Pool 1.2`[1] library in the `Apache Commons` collection, presented in Figure 2.1. In the `returnObject` method, first, the state of the shared object `pool`, is tested outside the synchronized block, by checking the value of the flag variable `isClosed`. If it is `true`, then some local computation occurs, followed by a synchronized block that dereferences the shared object `pool`. Method `close`, on the other hand, closes the pool by writing `null` to `pool` and setting `isClosed` to true, signaling that the pool has been closed.

The error in this code (and such errors are very typical) stems from the fact that the check of `isClosed` in method `returnObject` is not within the synchronized block; hence, if a thread executing the `returnObject` method performs the check at line 3, and then a concurrent thread executes the method `close` before the synchronized block begins, then the access to object `pool` at line 10 will raise an uncaught *null-pointer dereference exception*.

In a dynamic testing setting, consider the scenario where we observe an execution π with two threads T and T', where T executes the method `returnObject` first, and then, T' executes the method `close` after T finishes executing `returnObject`. There is no null-pointer

[1] http://commons.apache.org

```
1  public void returnObject(Object o){
2    ...
3    if (isClosed)
4        throw new PoolClosedEx();
5    synchronized (this) {
6        numActive--;
7        ...
8        ... = modCount;
9        ...
10       pool.push(o);
11   }
12 }
```

```
13 public void close(){
14   synchronized (this) {
15       ...
16       modCount = ...
17       ...
18       pool = null;
19       isClosed = true;
20   }
21 }
```

Figure 2.1: Code snippet of the buggy implementation of `Pool 1.2`.

dereference in the execution. Our goal is to predict a permutation of the events of π (called schedule) that causes a null-pointer dereference.

Our prediction for null-pointer dereferences works as follows. In the run π, thread T reads a *non-null* value from the shared object `pool` when the object method `pool` is called at line 10. Also, T' writes a *null* value to the same shared object `pool` at line 18. Our prediction approach identifies that read-write pair and searches for alternative schedules π' in which, the read at line 10 (in T) reads the value `null` written by the write at line 18 (in T'). Therefore, it predicts a run π' in which T is executed first until it gets to the synchronized block at line 5, followed by the execution of T' and then the execution of the synchronized block in T, which leads to a null-pointer dereference exception.

Our prediction algorithm observes accesses to shared variables and synchronization events but suppresses the semantics of the local computation entirely and does not even observe them. Then, it identifies *null-WR* pairs $\alpha = (e, f)$, where e is a write of null to a variable and f is a non-null read of the same variable in the observed run. Then, it encodes the problem of finding

a *sound* permutation of events of the observed run in which f is reading the null value written by e as a logical constraint system or an AI planning problem and uses the state-of-the-art SMT solvers and planners to search for an answer (if there exists any solution).

2.2 Preliminaries

Our prediction technique, similar to other prediction techniques, is based on program runs, i.e., a run of the program is observed and then the information available in the run is used for predicting other runs. Here, we first discuss what kind of information is available in program runs and then provide a background on the maximal causal model [74] which is the basis for our prediction technique.

2.2.1 Global Traces

We define a *global trace* to be a sequence of global computation (i.e., accesses to shared variables) and synchronization events. Note that a global trace does not contain any information about local computation (i.e., reads and writes to local variables) in the execution.

We assume a set of thread identifiers T= $\{T_1, T_2, ...\}$ and define a set of shared variables SV = $\{sv_1, sv_2, ...\}$ that the threads can access. Let $Init(x)$ and $Val(x)$ represent the initial value and the set of possible values that the shared variable $x \in SV$ can get, respectively. We also fix a set of global lock variables L.

The set of actions that a thread can perform on the set of shared variables SV and global locks L is defined as:

$$\begin{aligned}\Sigma = \quad & \{\mathit{rd}(x, val),\ \mathit{wt}(x, val) \mid x \in SV, val \in Val(x)\} \cup \\ & \{\mathit{ac}(l),\ \mathit{rel}(l) \mid l \in L\} \ \cup \ \{\mathit{tf}(T_i) \mid T_i \in T\}\end{aligned}$$

Actions $\mathit{rd}(x, val)$ and $\mathit{wt}(x, val)$ correspond to reading value val from and writing value val to shared variable x, respectively. Actions $\mathit{ac}(l)$ and $\mathit{rel}(l)$ represent acquiring and releasing

lock l, respectively. Action *tf*(T_i) represents the creation of thread T_i.

We denote the execution of an action by a thread as an *event*. Formally, an event is a tuple $(T_i, a) \in T \times \Sigma$. Let EV denote the set of all possible events. The sequence of events observed during an execution of a concurrent program forms a global trace:

Definition 2.2.1 (Global Trace)**.** *A global trace is a finite string $\pi \in EV^*$. By $\pi[n]$, we denote the n^{th} event of π. Given a global trace π, $\pi|_{T_i}$ is the projection of π to events involving T_i.*

In this chapter, whenever we refer to traces we mean global traces. A global trace π is *lock-valid* iff it respects the semantics of locking, i.e., two threads cannot obtain the same lock simultaneously.

Definition 2.2.2 (Lock-Valid Traces)**.** *Let π be a global trace and $\pi|_{T_i,l}$ be the projection of $\pi|_{T_i}$ on acquire and release events of lock l. Then, π is lock-valid iff*

(i) For each lock l, $\pi|_{T_i,l}$ (if it is not empty) starts with an acquire event $(T_i, ac(l))$ and acquire events $(T_i, ac(l))$ alternate with corresponding lock release events $(T_i, rel(l))$ in $\pi|_{T_i,l}$, and

(ii) For each acquire event $\pi[m] = (T_i, ac(l))$ either (1) there exists a corresponding release event $\pi[n] = (T_i, rel(l))$ such that $m < n$, and there are no acquire or release events of lock l by other threads between $\pi[m]$ and $\pi[n]$, or (2) the lock is not released by T_i in π (i.e, there is no event $(T_i, rel(l))$ after $\pi[m]$) and there are no acquire or release events of lock l by other threads after $\pi[m]$.

Let π be a lock-valid trace. *Lock-sets* and *lock acquisition histories* for π are defined as follows:

Definition 2.2.3 (Lock-Sets and Acquisition Histories (from [39]))**.** *Lock-Set(T_i, $\pi[j]$) is defined to be the set of locks acquired but not released by T_i before $\pi[j]$ in π. Then, for thread T_i and lock l such that $l \in$ Lock-Set(T_i, $\pi[n]$), where n is the length of π, we define AH(T_i, l, π) be*

the set of locks that were acquired (and possibly released) by T_i after the last $(T_i, ac(l))$ event in π.

A global trace π is *data-valid* iff it respects the read-write constraints, i.e., each read from a shared variable should read the value written by the most recent write event to that shared variable.

Definition 2.2.4 (Data-Valid Traces)**.** *Let π be a global trace. Then, π is data-valid iff for each n such that $\pi[n] = (T_i, rd(x, val))$, either*

(i) The last write event to x writes value val; i.e., there is m such that $m < n$ and $\pi[m] = (T_i, wt(x, val))$ and there is no k such that $m < k < n$ and $\pi[k] = (T_q, wt(x, val'))$ for any val' and any thread T_q, or

(ii) There is no write event to variable x before the read and val is the initial value of x; i.e., there is no m such that $m < n$ and $\pi[m] = (T_j, wt(x, val'))$ (for any val' and any thread T_j), and $val = Init(x)$.

A global trace π is *creation-valid* iff every thread is created at most once and the events of each thread occur after it is created.

Definition 2.2.5 (Creation-Valid Traces)**.** *A global trace π is creation-valid iff for every $T_i \in T$, there is at most one event of the form $(T_j, tf(T_i))$ in π, and, if such an event exists, then all events in π_{T_i} happen after this event in π.*

Each global trace obtained from an execution of a program defines a total order on the set of events in it. Furthermore, there is an induced partial order between the events of each thread:

Definition 2.2.6 (Program Order)**.** *Let π be a global trace obtained from an execution of a program. We define a partial relation $\sqsubseteq_k$ such that $\pi[i] \sqsubseteq_k \pi[j]$ iff $\pi[i], \pi[j] \in \pi|_{T_k}$, and $i \leq j$. The union of these partial orders $\cup_{T_i \in T} \sqsubseteq_i$ is referred to as* program order.

The program order arranges the events in each thread according to their order in π.

Definition 2.2.7 (Causal Relation)**.** *Let π be a global trace. We define a partial order $\preceq$ on the set of events in π such that $\pi[i] \preceq \pi[j]$ iff $i \leq j$ and*

(i) *$\pi[i]$ and $\pi[j]$ are performed by the same thread, or*

(ii) *$\pi[i] = (T_p, ac(l))$, $\pi[j] = (T_q, rel(l))$, and $T_p \neq T_q$, or*

(iii) *$\pi[i] = (T_p, rel(l))$, $\pi[j] = (T_q, ac(l))$, and $T_p \neq T_q$, or*

(iv) *$\pi[i] = (T_p, wt(x, val))$ and $\pi[j] = (T_q, rd(x, val'))$ or $\pi[j] = (T_q, wt(x, val'))$ for some shared variable x and values val and val', and $T_p \neq T_q$, or*

(v) *$\pi[i] = (T_p, rd(x, val))$, $\pi[j] = (T_q, wt(x, val'))$ for some x, val, val', and $T_p \neq T_q$, or*

(vi) *$\pi[i] = (T_p, tf(T_q))$ and $\pi[j]$ is performed by T_q (i.e., $\pi[j] = (T_q, -)$), and $T_p \neq T_q$.*

Transitive closure of $\preceq$ (represented by $\preceq^*$) is called causal relation.

The causal relation, for each event in the trace, defines a set of events on which the event depends.

2.2.2 Maximal Causal Model of Prediction

Our prediction technique is based on the maximal causal model [74]. The maximal causal model works at the shared communication level, i.e., it only considers accesses to shared variables and synchronization events. Given a global trace of a concurrent program, a causal model is obtained which is both sound and maximal; i.e., all traces consistent with the causal model correspond to feasible executions of the concurrent program under analysis, and assuming only the global trace and no knowledge about the source code of the program, the model captures more feasible executions than any other sound causal model. In the following, we define the maximal causal model as a set of *precisely predictable* runs.

Definition 2.2.8 (Precisely Predictable Runs (adapted from [74]))**.** *Let π be a global trace over a set of threads T, shared variables SV, and locks L, obtained from a program execution. Global trace π' is* precisely predictable *from π if*

(i) for each $T_i \in T$, $\pi'|_{T_i}$ is a prefix of $\pi|_{T_i}$,

(ii) π' is lock-valid,

(iii) data-valid, and

(iv) creation-valid.

Let $PrPred(\pi)$ denote the set of all runs with global traces that are precisely predictable from π, called precisely predictable runs from π.

The first condition above ensures that the sequence of events of T_i occurred in π' is a prefix of the sequence of events of T_i occurred in π. Note that we are forcing the thread T_i to read the same values of shared variables as it did in the original run. Along with data-validity, this ensures that the thread T_i reads precisely the same values and updates the local state in the same way as in the observed run. Lock-validity and creation-validity are, of course, required for feasibility. The following theorem states the soundness of the prediction that guarantees all predicted runs to be feasible:

Theorem 2.2.9 (from [74])**.** *Let P be a program and π be a global trace corresponding to an execution of P. Then, every precisely predictable run in $PrPred(\pi)$ is a feasible of P.*

The complete proof of this theorem can be found in [74]. The intuition behind why the theorem holds is that as long as each read event reads in the same value as it did in the observed run, each thread is forced to take the same path as it took in the observed run. Since the observed run is a feasible run, all predicted runs are guaranteed to be feasible.

2.3 Predicting Null-Pointer Dereferences

Although null-pointer dereferences could occur on both local and shared variables, a prediction that takes into account local variables and local computation encounters scalability issues for large program runs. We propose an approximation to null-pointer dereference prediction that works at the shared communication level, i.e., accessing to shared variables and synchronization events. Consider a thread T that in some interleaving reads a *non-null* value from shared variable x and subsequently does some computation locally using the non-null value, and consider the task of predicting whether this could result in a null-pointer dereference. Our approximation of the prediction problem at the shared communication level asks for a run that forces the thread T to read a *null* value from x. Note that this approximation is neither sound nor complete: Thread T may read null for x but may not dereference the pointer (e.g., it could check if x is null), and there may be runs where the value read is not null and yet the local computation causes a null-pointer dereference. However, such an approximation is absolutely necessary to scale to large runs, as it is imperative that local computation is not modeled. To guarantee the feasibility of the predicted runs, our prediction approach is based on the maximal causality model (discussed in Section 2.2.2). Now, we formally define the precise prediction problem for forcing null-reads.

Definition 2.3.1 (Precisely Predictable Null-Reads)**.** *Let π be a global trace obtained from an execution of a program P. We say that a run with global trace π' is a precisely predictable run from π that forces null-reads if there is a thread T_i and a variable x such:*
(i) $\pi' = \pi''.f$ where f is of the form rd$(x, null)$ and π'' is a precisely predictable from π (see Definition 2.2.8), and
(ii) there is some val $\neq$ null such that $(\pi''|_{T_i})$.rd(x, val) is a prefix of $\pi|_{T_i}$.

Intuitively, the above conditions require that the π' be a precisely predictable from π followed by a read of null by a thread T_i on variable x, and further, in the observed trace π, thread T_i must be performing a non-null read of variable x after performing its events in π''. The

above captures the fact that we want a precisely predictable run followed by a single null-read that corresponds to a non-null read in the original observed run. Note that π' itself is not in $PrPred(\pi)$, but is always feasible in the program P.

The first step of our prediction is to identify a set of *null-WR* pairs $\alpha = (e, f)$, where e is a write of null to a variable and f is a non-null read of the same variable, in the observed trace. Then, we perform a static lock-based analysis according to the *null-WR* pairs to identify a small segment of the observed run on which the prediction can focus. Finally, we encode the prediction problem as an SMT problem or an AI planning problem to find an answer. In the following, we discuss how we identify the *null-WR* pairs and then present the static pruning analysis. The encodings are presented in Sections 2.4 and 2.5.

2.3.1 Identifying *null-WR Pairs* Using Lock-based Analysis

Each *null-WR* pair $\alpha = (e, f)$ is a tuple where e is a write of null to a shared variable x and f is a non-null read of the same shared variable. We would like to identify pairs that are feasible at least according to the hard constraints of thread-creation and locking in the program. For instance, if a thread writes to a shared variable x and reads from it in the same lock-protected region of code, then clearly the read cannot match a write protected by the same lock in another thread. Similarly, if a thread initializes a variable x to a non-null and then creates another thread that reads x, clearly the read cannot see the uninitialized x. We use a lock-based static analysis of the run (without using a constraint solver) to filter out such infeasible *null-WR* pairs .

The idea is to check if for a *null-WR* pair $\alpha = (e, f)$, f can read from e in a (not necessarily feasible) run that only respects lock-validity and creation-validity constraints (and not data-validity). Creation validity is captured by computing a causal relation among the threads (Definition 2.2.7) by considering only the program order and thread creation constraints (i.e., items (i) and (vi)) in Definition 2.2.7. If $f \preceq^* e$ according to this relation, then clearly f cannot occur after e and the *null-WR* pair should be discarded. Lock-validity is captured by reducing the problem of realizing the pair (e, f) to *pairwise reachability* under nested locking [39],

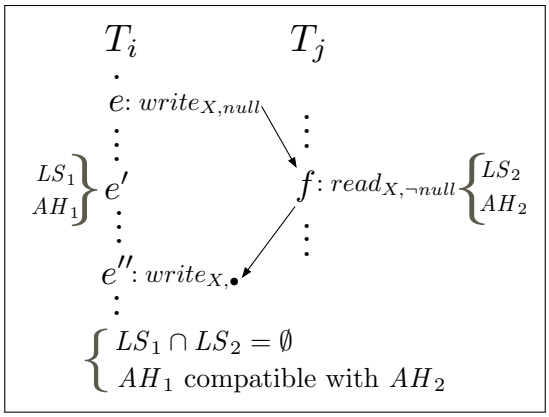


Figure 2.2: Static lock-based analysis for feasibility of a *null-WR* pair $\alpha = (e, f)$.

which is then solved by computing lock-sets and acquisition histories for each event. Similar techniques have been exploited for finding atomicity violations in the tool PENELOPE [82]. Here, we briefly discuss the reduction for lock-validity checking.

Consider an observed trace π and a *null-WR* pair $\alpha = (e, f)$ where f (a read in thread T_j) occurs before e (a write in thread T_i) in π. Let us assume that e'' is the next write event (to the same variable accessed in e and f) in T_i after e.

We claim that if there exists a lock-valid run with global trace π' (obtained by permuting the events in π) in which f reads the null value provided by e, then in π', f should be scheduled after e, but before e''; if f is scheduled before e, then it would not read from e. If f is scheduled after e'' then the write in e'' overwrites the null value written by e before it reaches f. This means that there should exist an event e' of thread T_i, occurring between events e and e'', that is right before (or after) f in π'; in other words, e' and f are co-reachable. Note that in cases that there is no write event (to the same variable accessed in e and f) in T_i after e, e' could be any event in T_i after e.

As shown in Figure 2.2, we iterate over all possible events e' of T_i between e and e'' in π and use a simple technique [39] to check the co-reachability of e' and f. As proposed in [39], the co-reachability check is done by examining the lock-sets and acquisition histories (See Definition 2.2.3) at e' and f: The lock-sets at e' and f must be disjoint and the acquisition

histories at e' and f must be compatible, i.e., there are no locks $l \in$ Lock-Set(T_i, e') and $l' \in$ Lock-Set(T_j, f) with $l' \in$ AH(T_i, l, π) and $l \in$ AH(T_j, l', π).

2.3.2 Static Pruning

Given a global trace π, we collect all *null-WR* pairs $\alpha = (e, f)$ as discussed in the previous section. Then, according to the prediction problem for each *null-WR* pair $\alpha = (e, f)$, we have to search for a lock-valid, data-valid and creation-valid trace, consisting of the events in π, in which f is reading the null value written by e. However, instead of using the whole π (which can be very large) for the purpose of prediction, we slice a *relevant* segment of it, and use the segment instead. This segment is often orders of magnitude smaller than π itself, and hence the scalability of prediction is increased. However, any run predicted from the segment will still be feasible. While this limits the number of predictable runs in theory, it does not prevent us from finding errors in practice (in particular, *no error* was missed due to pruning in our experiments).

Consider a global trace π and a *null-WR* pair $\alpha = (e, f)$. The idea behind pruning is to first prune away a set of events in π which are *not causally before* e or f as they play no role in occurrence of e or f. Assume that π' is the new trace obtained after pruning. Then, in the next step we find the largest prefix of π' before reaching e and f such that all of the locks are free at the end of this prefix. The intuition behind this is that such a prefix can be replayed in the predicted run precisely in the same way as it occurred in the observed run. The prediction problem is then restricted to the remained suffix, containing e and f, while the initial values of shared variables for prediction is obtained by the last write to each shared variable in the prefix segment. The prefix then can be stitched to a run predicted from the suffix since the suffix will start executing from a state where no locks are held.

For the first step, let ρ_α define the *smallest* subset of events of π that satisfies the following properties: (1) ρ_α contains events e and f, (2) for any event e' in ρ_α, all events e'' that are causally before it (i.e., $e'' \preceq^* e'$ according to Definition 2.2.7) are in ρ_α, and (3) for every event

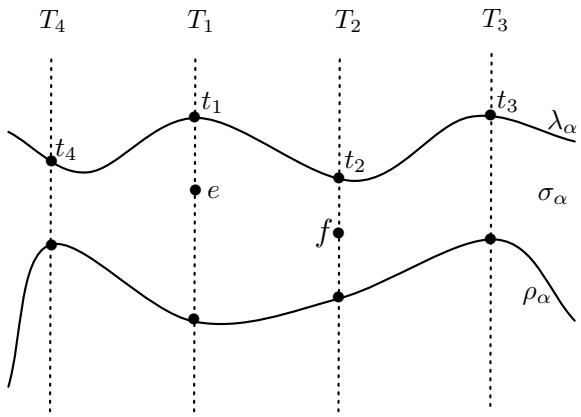


Figure 2.3: Static pruning according to a *null-WR* pair $\alpha = (e, f)$.

corresponding to a lock acquire in ρ_α, its corresponding release event is also in ρ_α.

The intuition is that events that are not in ρ_α are not relevant for the scheduling of the *null-WR* pair. Figure 2.3 presents a run of a program with 4 threads that is projected into individual threads. Here, e belongs to thread T_1 and f belongs to thread T_2. The cut labeled ρ_α marks the boundary after which all events are not *causally before* e or f, and hence, need not be considered for the generation of the new run.

For the second step, we identify a causally prefix-closed set of events before e and f to remove. For the *null-WR* pair α, let λ_α define the *largest* subset of events of ρ_α that has the following properties: (1) it does not contain e or f, (2) for any event e' in λ_α, all events e'' that are causally before it (i.e., $e'' \preceq^* e'$) are in λ_α, and (3) for any event e' in T_i such that e' is the last event of T_i in λ_α, the Lock-Set(T_i, e') is empty. In the figure, the curve labeled λ_α marks the boundary of λ_α where events $T_1, \ldots, T_4$ have empty lock-sets.

The run segment relevant to a *null-WR* pair α is then defined as the set of events in $\sigma_\alpha = \rho_\alpha \setminus \lambda_\alpha$ scheduled according to the total order in π. This run segment is passed to the run prediction phase, in the place of the whole run π.

2.4 Encoding as an SMT Problem

In this section, we encode the problem of precisely predicting a run that realizes a *null-WR* pair as a set of logical constraints. More specifically, given a trace π and a *null-WR* pair $\alpha = (e, f)$, we encode the set of all possible runs in the maximal causal model that forces the read f to read the null value written by e as a set of constraints and use a constraint solver to search for a solution. The constraints fall into the *Difference Logic* [52] which is efficiently decidable [41].

Prediction according to the maximal causal model is basically an encoding of the creation-validity, data-validity, and lock-validity constraints using logic, where quantification is removed by expanding over the finite set of events under consideration. Modeling this using constraint solvers has been done before ([68]) in the context of finding data races. We reformulate this encoding briefly here (in Section 2.4.1) and adapt it to predict null-pointer dereferences. We also propose a wide set of carefully chosen optimizations on this encoding.

Prediction based on the maximal causal model is sound, in the sense that it guarantees feasibility of the predicted runs. However, sound prediction under the maximal causal model can be too restrictive in some cases, i.e., the constraint system is unsatisfiable. Slightly diverging from the maximal causal model can lead to prediction of runs that are also feasible in the original program in many cases. In Section 2.4.2, we present a relaxation technique on the maximal causal model and show how the precise encoding is changed to reflect the relaxation technique.

2.4.1 Precise Prediction

In this section, first we present how the maximal causal model can be captured by logical constraints. Then, we show how these constraints are adapted for predicting runs realizing a specific *null-WR* pair. Finally, we provide a set of optimizations that reduce the size of the constraint system soundly.

ψ: $\quad PO \wedge FC \wedge LC \wedge DC$

PO: $\quad \bigwedge_{T_i \in T} PO_{T_i} \wedge C_{init}$

C_{init}: $\quad \bigwedge_{i=1}^{n} (t_{init} < t_{e_{i,1}})$

PO_{T_i}: $\quad \bigwedge_{j=1}^{m_i - 1} (t_{e_{i,j}} < t_{e_{i,j+1}})$

FC: $\quad \bigwedge_{T_i \in T} (t_{e_{tf(T_i)}} < t_{e_{i,1}})$

LC: $\quad LC_1 \wedge LC_2$

LC_1: $\quad \bigwedge_{T_i \neq T_j} \bigwedge_{l \in L} \bigwedge_{\substack{[aq,rl] \in L_{T_i,l} \\ [aq',rl'] \in L_{T_j,l}}} (t_{rl} < t_{aq'} \vee t_{rl'} < t_{aq})$

LC_2: $\quad \bigwedge_{T_i \neq T_j} \bigwedge_{l \in L} \bigwedge_{\substack{aq \in NoRel_{T_i,l} \\ [aq',rl'] \in L_{T_j,l}}} (t_{rl'} < t_{aq})$

DC: $\quad \bigwedge_{x \in SV} \bigwedge_{val \in Val(x)} \bigwedge_{r \in R_{x,val}} (\bigvee_{w' \in W_{x,val}} Coupled(r, w'))$

$Coupled(r, w)$: $\quad (t_w < t_r) \wedge \bigwedge_{w' \in W_x \setminus \{w\}} ((t_r < t_{w'}) \vee (t_{w'} < t_w))$

Figure 2.4: Constraint system capturing the maximal causal model.

Capturing Maximal Causal Model Using Logic

Given a trace π, we first encode the constraints on all runs precisely predictable from it, using the maximal causal model, independent of the specification that we want runs that realize a given *null-WR* pair. A predicted run can be seen as a total ordering of the set of events E in the trace π. We use an integer variable t_e to encode the *timestamp* of an event $e \in E$ in the predicted run. Using these timestamps, we logically model the constraints required for precisely predictable runs (see Definition 2.2.8); i.e., the runs should respect the program order implied by π, and should be lock-valid, data-valid, and creation-valid.

Figure 2.4 illustrates the various constraints. The constraint system is a conjunction of program order constraints (*PO*), creation-validity constraints (*FC*), data-validity constraints (*DC*), and lock-validity constraints (*LC*).

Suppose that the given trace π consists of the events of n different threads, and let $\pi|_{T_i}$ =

$e_{i,1}$, $e_{i,2}$, ..., e_{i,m_i} be the sequence of events in π that relates to thread T_i.

PO: The program order constraint (*PO*) captures the condition that the predicted run respect the program order of the observed run. We consider an initial event e_{init} which corresponds to the initialization of shared variables. This event should happen before any thread starts the execution in any predicted run; constraint C_{init} encodes this fact. The constraint PO_i requires that the predicted run obey the order of events in thread T_i, and *PO* requires that all threads meet their program order.

FC: Turning to creation-validity, suppose that $e_{tf(T_i)}$ is the event that creates thread T_i. Then, the constraint *FC* requires that the first event of T_i can only happen after $e_{tf(T_i)}$. Combined with program order constraint, this means that all events before the creation of T_i in the thread that created T_i must also occur before the first event of T_i in the predicted run.

$\boldsymbol{LC}_1 \wedge \boldsymbol{LC}_2$: Lock-validity (see Definition 2.2.2) is captured by the formula *LC*. We assume that each lock acquire event *aq* of lock l in the observed run is matched by precisely one lock release event *rl* of lock l in the same thread, unless the lock is not released by the thread in the run. Each lock acquire event *aq* and its corresponding lock release event *rl* define a lock block, represented by $[aq, rl]$. Let $L_{T_i,l}$ be the set of lock blocks in thread T_i regarding lock l. Then, LC_1 ensures that no two threads can be inside lock blocks of the same lock l, simultaneously. Turning to locks that never get released, the constraint LC_2 ensures that the acquire of lock l by a thread that never releases it must always occur after all releases of lock l in other threads. In this formula, $NoRel_{T_i,l}$ stands for lock acquire events in T_i with no corresponding lock release event.

DC: The data-validity constraints *DC* (see Definition 2.2.4) capture the fact that reads must be coupled with appropriate writes; more precisely, that every read of a value from a variable must have a write before it writing that value to that variable, and moreover, there is no other intermediate write to that variable. Let $R_{x,val}$ represent the set of all read events that read value val from variable x in π, W_x represent the set of all write events to variable x, and $W_{x,val}$ represent the set of all write events that specifically write value val to variable x. For each

read event $r = \textit{rd}(x, val)$ and write event $w \in W_{x,val}$, the formula $\textit{Coupled}(r, w)$ requires that in the predicted run, w should happen before r and all other writes to variable x should either happen before w or after r; i.e., w should be the most recent write to variable x before r and hence r is coupled with w. The constraint *DC* requires that each read be coupled with a write that writes the same value as the read reads in the observed run.

Predicting Runs for a *null-WR Pair*

Now, we present how the constraint system proposed above can be adapted for predicting runs (consistent with the maximal causal model) that realize a *null-WR* pair. Suppose that $\alpha = (e, f)$ is the *null-WR* pair and π is (the segment of) the observed trace containing e and f which is considered for prediction. Notice that in the observed run f reads a non-null value while we will force it to read null in the predicted run by coupling it with write event e. Therefore, we drop from the data-validity formula (DC) that the value read at f should be the same as in the observed run. In addition, we need to add a constraint NC = $\textit{Coupled}(f, e)$ that forces the read f be coupled with the write e, i.e., e occurs before f while avoiding any other write to the corresponding variable between e and f.

Suppose that f is performed by thread T_i and e is performed by thread T_j. Since both e and f should occur in the predicted run, according to the program order, all of the events in T_i and T_j before f and e should appear in the predicted run. Note that once f reads a different value, we no longer have any predictive power on what the program will do (as we do not examine the code of the program but only its runs). Consequently, we cannot predict any events causally later than f, i.e., f should be the last event in the predicted run.

A further complication is how to deal with events that are after e in T_j and events in threads other than T_i and T_j. Note that some of these events may need to occur in order to satisfy the requirements of events before f in the predicted run (for instance a read before f may require a write after e to occur). However, not all of these events are required to happen before f. Our strategy is to let the solver figure out the precise set of events that are required in the predicted

run. Therefore, the lock-validity and data-validity constraints are enforced on events that are scheduled *before* f (i.e., their timestamp is less than the timestamp of f). More precisely, we replace:

(i) $(\bigvee_{w' \in W_{x,val}} \mathit{Coupled}(r, w'))$ in the formula *DC* with $(\bigvee_{w' \in W_{x,val}} (t_r < t_f \wedge t_{w'} < t_f) \Rightarrow \mathit{Coupled}(r, w'))$,

(ii) $(t_{rl'} < t_{aq})$ in LC_2 with $(((t_{rl'} < t_f) \wedge (t_{aq} < t_f)) \Rightarrow (t_{rl'} < t_{aq}))$,

(iii) $(t_{rl} < t_{aq'} \vee\ t_{rl'} < t_{aq})$ in LC_1 with $((t_{aq'} < t_f) \wedge (t_{aq} < t_f)) \Rightarrow (t_{rl} < t_{aq'} \vee\ t_{rl'} < t_{aq}))$,

and add $\bigwedge_{T_i \in T} \bigwedge_{l \in L} \bigwedge_{[aq,rl] \in L_{T_i,l}} (t_{aq} < t_f \Rightarrow t_{rl} < t_f)$ to the constraint system to ensure that each lock block is completely scheduled before f if its lock acquire event is scheduled before f. This is to avoid introducing new non-released locks in the predicted runs that might lead to deadlock.

Optimizations

The data-validity constraint (*DC*) is expensive to express, as it is in Figure 2.4; in the worst case, it is cubic in the maximum number of accesses to any variable. There are several optimizations that reduce the number of constraints in the encoding. Suppose that $r = (T_i, \mathit{rd}(x, val))$ is a read from shared variable x. Then,

(i) each write event w' to x that occurs after r in T_i, i.e., $r \sqsubseteq_i w'$, can be excluded in the constraints related to coupling r with a write in constraint *DC*.

(ii) suppose that w is the most recent write to x before r in T_i. Then, each write event w' before w in T_i, i.e., $w' \sqsubseteq_i w$, can be excluded in the constraints related to coupling r with a write in constraint *DC*.

(iii) when r is being coupled with $w \in W_{x,val}$ in thread T_j, each write event w' before w in T_j, i.e., $w' \sqsubseteq_j w$, can be excluded as candidates for e'' in the formula $\mathit{Coupled}_{r,w}$.

(iv) suppose that r is being coupled with $w \in W_{x,val}$ in thread T_j and w' is the next write event to x after w in thread T_j. Then, each write event w'' after w' in T_j, i.e., $w' \sqsubseteq_j w''$, can be excluded as candidates for e'' in the formula $\mathit{Coupled}_{r,w}$.

(v) event r can be coupled with e_{init} only when there is no other write event to x before r in T_i, i.e., $\nexists w.\ (w \sqsubseteq_i r \wedge w \in W_x)$. Furthermore, it is enough to check that the first write event to x in each thread (if it exists) is performed after r.

The lock-validity constraint (LC), which is quadratic in the number of lock blocks, is also quite expensive in practice. We optimize the constraints as follows:

(i) If a read event r in thread T_i can be coupled with only *one* write event w which is in thread T_j then in all precisely predictable runs, w should happen before r. Therefore, the lock blocks according to each lock l that are in T_j before w and the lock blocks according to lock l that are in T_i after r are already ordered. Hence, there is no need to consider constraints preventing T_i and T_j to be simultaneously in such lock blocks in LC_1. In practice, this greatly reduces the number of constraints.

(ii) When considering lock acquire events with no corresponding release events in LC_2, it is sufficient to only consider the *last* corresponding lock blocks in each thread and exclude the earlier ones from the constraint.

We present a simple example of a null-pointer dereference prediction problem in Appendix A and provide the logical constraint encoding of the problem based on the encoding presented in this section.

2.4.2 Relaxed Prediction

To guarantee feasibility of predicted runs, the encoding based on the maximal causal model restricts all of the reads in the predicted run (except f) to read the same value as they did in the

observed trace. However, this can be too restrictive in the sense that no run can be predicted with this restriction. For instance, suppose that $\alpha = (e, f)$ is a *null-WR* pair where e is in thread T_i and f is in thread T_j. Furthermore, suppose that in the observed trace π, all events of T_j occur before events of T_i. In this case, if there is a read event r in thread T_i before e ($r \sqsubseteq_i e$) that can be matched only with a write w in T_j after f ($f \sqsubseteq_j w$) then there is no precisely predictable run in which α is realized. However, if the value being read by r does not affect the paths taken by the threads (for example, there is no conditional that checks the value of this variable), ignoring the constraints related to r will help us in finding a feasible run.

We hence have a trade-off between two choices; we would like to maintain the same values read for as many shared variable reads as possible to increase the probability of getting a feasible run, but at the same time allow a few reads to read different values to make it possible to predict some runs. Our proposal is an iterative algorithm for finding the *minimum* number of reads that can be exempt from data-validity constraints that will allow the prediction algorithm to find *at least* one run. We define a suitable *relaxed* logical constraint system to predict such a run. Our experiments show that exempting a few reads from data-validity constraints greatly improves the flexibility of the constraints and increases the possibility of predicting a run, and at the same time, the predicted runs are often feasible.

Suppose that there are n read events in trace π. The iterative algorithm works as follows: The data-validity constraints are expressed so that we specifically ask for n reads to be coupled precisely. If we fail to find a solution then we attempt to find a solution that couples $n - 1$ reads precisely in the next round. We keep decrementing n and searching for a solution until the constraint system becomes satisfiable and a run (solution) is found or a threshold for the number of relaxed reads is reached. The changes required in the encoding to make this possible are described below.

For every read event $r_i \in R$, we introduce a new Boolean variable, b_i, that is true if the data-validity constraint for r_i is satisfied, and false, otherwise. In addition, we consider an integer variable $bInt_i$ which is 0 if b_i is false, and 1 if b_i is true. This is enforced through a set

of constraints, one for each $r_i \in R$: $[(b_i \Rightarrow bInt_i = 1) \wedge (\neg b_i \Rightarrow bInt_i = 0)]$. Furthermore, for each $r_i \in R$, in the *DC* constraint, we change the sub-term $(\bigvee_{w' \in W_{x,val}}(t_r < t_f \wedge t_{w'} < t_f) \Rightarrow \textit{Coupled}(r, w'))$ to $(b_i \Rightarrow (\bigvee_{w' \in W_{x,val}}(t_r < t_f \wedge t_{w'} < t_f) \Rightarrow \textit{Coupled}(r, w')))$, forcing the data-validity constraint for read r_i to hold when b_i is true. Note that with these changes, we require a different theory, i.e., *Linear Arithmetic* in the SMT solver to solve the constraints, compared to the Difference Logic which was used for our original set of constraints.

Initially, we set a threshold η to be $|R|$, i.e., the number of all read events. In each iteration, we assert the constraint $\sum_{1 \leq i \leq |R|} bInt_i = \eta$, which specifies the number (η) of data-validity constraints that should hold in that iteration. If no run can be predicted with the current threshold η (i.e., the constraint solver reports unsatisfiability), then η is decremented in each iteration, until the formula is satisfiable. This way, when a satisfying assignment is found, it is guaranteed to have the maximum number of reads that respect data-validity possible for predictable run. Note that once $\eta < |R|$, the predicted run is not theoretically guaranteed to be a feasible run. However, in practice, η is close to $|R|$ and predicted runs are usually feasible in the program.

2.5 Encoding based on AI Automated Planning Techniques

AI automated planning is a rich and rapidly evolving area of research [56]. The last 15 years have seen tremendous advances in the field with the development of compact encoding techniques for state representation and transition functions, together with highly effective search techniques based on both satisfiability (SAT) and heuristic search. These advances have not only led to the development of fast and highly effective AI planning systems, but they have also led to advances in model checking [23, 26, 28] and related fields. Therefore, we elected to explore the effectiveness of AI automated planning as a vehicle for prediction. In this context, the prediction of a run realizing a *null-WR* pair is characterized as a sequential planning problem with the temporally extended goal of achieving the particular violation being sought.

In this section, we propose a means of encoding the realization of a *null-WR* pair as a temporally extended goal and characterizing the overall task as a classical planning problem using the maximal causal model (discussed in Section 2.2.2). Despite the focus on null reads, it is important to note that a very similar encoding can be applied to prediction of runs containing other types of concurrency violations like atomicity violations and data races. Indeed, the merit of using AI planning is in exploiting the rich compact encodings of transition systems that planners use, the ability to encode complex violation patterns (at least anything that can be encoded in Linear Temporal Logic [60, 87]) as planning goals and the highly optimized heuristic search techniques that have been honed over the past decade. Here, we only focus on the encoding of precise prediction, using the maximal causal model. Note that our precise prediction technique does not require incorporating any numerical data. We leave the investigation of possible encodings of prediction based on the relaxation technique for future work.

In the rest of this section, we first present a background on planning. Then, we show how the prediction problem can be encoded as a planning problem.

2.5.1 Background on Planning

Informally, a planning problem can be described as follows: *given a description of a set of actions an agent can perform, together with a specification of an initial state and a goal, the task of automated planning is to generate a set of actions, together with some ordering constraints, such that if those actions are executed by an agent, starting in the initial state, following the ordering constraints, they will lead to a state in which the goal is achieved.* Classical planning problems are characterized by a finite initial state that is completely specified, a finite set of actions that are deterministic, and a goal condition that is restricted to conditions placed on the final state of the system. More formally, a STRIPS classical planning problem [56] is defined as a tuple $P = (S_0,\ F,\ A,\ G)$ where F is a finite set of atomic facts, $S_0 \subseteq F$ is the *initial state*, $G \subseteq F$ specifies a set of goal states where the facts comprising G hold, and A is a finite set of deterministic *actions*. Each action $a \in A$ is described by a tuple ($pre(a)$, $add(a)$,

$del(a)$) where $pre(a)$ is a pair ($pre^+(a)$, $pre^-(a)$) of disjoint subsets of F that define, respectively, the positive and negative preconditions of action a, respectively. (For STRIPS classical planning, $pre^-(a)$ is empty.) Further, $add(a)$ and $del(a)$ are disjoint subsets of F that define, respectively, the positive and negative effects of action a.

Here, a *planning state* is a subset of elements in F. Classical planning assumes complete information about the planning state. Therefore, every $f \in F$ that is not explicitly mentioned in a planning state, including the initial state, is assumed to be false in that state. Action a is applicable in a planning state $s \subseteq F$ iff $pre^+(a) \subseteq s$ and $pre^-(a) \cap s = \emptyset$. Applying action a in state s would result in a new, successor state, $succ(a,\ s) = (s \backslash del(a)) \cup add(a)$. The goal G corresponds to a set of planning states and a $plan$, (strictly speaking a *sequential* plan, henceforth just "plan") $\pi = \overrightarrow{a}$, consists of a finite sequence of actions $a_0, ..., a_n$ which, when applied to the initial state, will produce a state in G.

Much of the research and many of the advances in automated planning have been with respect to classical planning. However, many real-world planning problems do not fit within this narrow characterization. One such restriction is that the goal to be achieved by an agent – the objective of the search – is restricted to some property of the *final* state of the system. A relaxation of this restriction is to support goals that are *temporally extended*, i.e., where the objective can specify properties that occur along the trajectory of states realized by a plan execution. In the spirit of this, a *temporally extended planning problem*, P (e.g., [1]), in this setting is a classical planning problem $P = (S_0,\ F,\ A,\ G)$ where the goal G is not restricted to a final-state goal, but rather is a set of facts together with some ordering constraints. Such *temporally extended goals* are often specified in linear temporal logic (LTL) [60]. A sequential plan for a temporally extended goal G is simply a sequence of actions, $\pi = \overrightarrow{a}$, which when applied to the initial state results in a sequence of states that entails G.

Automated planning problems are typically encoded in terms of a *planning domain description* that describes the dynamics of the planning problem – parameterized representations of the actions, their preconditions and effects, and by a *problem instance* that includes a de-

scription of the initial state and the goal. The de facto standard for specifying planning domains and planning instances is PDDL, the Planning Domain Definition Language [51]. PDDL has evolved over the years to address increasing needs for expressiveness, and is firmly established as the input specification language for most automated planning systems. PDDL3 [22], a recent version of PDDL, allows for the specification of temporally extended constraints and preferences in a subset of LTL.

Automated planning systems themselves vary in their approaches to plan generation. Two popular approaches are those based on heuristic search, as exemplified by the very successful Fast-Forward (FF) domain independent planning systems used here [31], and those based on SAT (e.g., [67, 40]). While these systems take PDDL as input, most transform the PDDL into an internal representation that is tailored to the needs of their search algorithm. Recent advances in automated planning have seen the development of effective planning techniques for a diversity of planning problems including cost-optimal planning, preference-based planning, net-benefit planning, and planning with nondeterministic effects of actions. These advances present further opportunities for the exploitation of planning in test generation.

2.5.2 Precise Prediction

We encode the dynamics of the given trace as an initial state S_0, a set of facts F, and a set of actions A. Actions correspond to events within the given trace. The facts record which actions (i.e., run events) have been executed and some specific properties relating to the most recent write to each shared variable and lock availability. The preconditions and effects for individual actions are written so as to enforce program order, and also to enforce lock-validity, data-validity and creation-validity that ensure that any plan generated from this planning instance corresponds to a precisely predictable run from the given global trace that guarantees feasibility. We illustrate this encoding in this section in detail.

We treat each *null-WR* pair $\alpha = (e, f)$ as a *temporally extended goal*. More specifically, it can be specified as an LTL formula *eventually (*$Happened_e$ *and next eventually (*$Happened_f$*))*

where $Happened_e$ and $Happened_f$ encode the occurrence of events e and f, respectively. As such, the task of predicting a run that realizes α is viewed as the automated generation of a plan with a temporally extended goal. Exploiting results proposed in [1] such problem can be transformed into a classical planning problem, by exploiting an established correspondence between LTL and Büchi automata [81]. In more restrictive cases, such as realizing a *null-WR* pair described here, there is an even simpler transformation of the temporally extended goals into a final-state goals [29] via what is effectively precondition control on the actions. We elaborate on this transformation at the end of this section.

Here, we present schemas or templates for the general PDDL encoding we employ for prediction. For ease of explanation, syntax does not strictly conform to PDDL syntax but is expressively equivalent. First, we illustrate how the maximal causal model is encoded in a planning domain such that any sequence of applicable actions correspond to a feasible execution of the corresponding program. Then, we describe how null reads, considered as temporary extended goals, can be compiled into constraints on the evaluation of the domain.

Capturing Maximal Causal Model Using Planning

Given a trace π, we use the maximal causal model to encode the set of precisely predictable runs from π (Definition 2.2.8).

Events and Program Order: We encode each event in π as an action in the planning domain. Therefore, we may have five different types of actions: read, write, thread creation, lock acquire, and lock release actions. Let $\pi|_{T_i} = \{e_{i,1}, e_{i,2}, ..., e_{i,m}\}$ be the projection of π on thread T_i. Suppose that we have $e_{i,1} \sqsubseteq_i e_{i,2}, e_{i,2} \sqsubseteq_i e_{i,3}, ..., e_{i,m-1} \sqsubseteq_i e_{i,m}$. According to the program order, event $e_{i,j+1}$ cannot occur before event $e_{i,j}$. Let action $Ac_{i,j}$ represent event $e_{i,j}$, i.e., the j^{th} event in thread T_i. For each action $Ac_{i,j}$ we introduce a predicate $(Done_{i,j})$ that indicates whether the action has been applied or not. These predicates are initially false and become true after the application of action $Ac_{i,j}$. To enforce program order, action $Ac_{i,j}$ requires $(Done_{i,(j-1)})$ to be true as a precondition.

In a general planning problem, an action may be applied several times to find a solution for the planning problem. Note that this cannot happen in our case as each action represents an event which can occur at most once in any run. Therefore, each action cannot be applied more than once in any plan. To encode this fact, predicate $(\text{NOT } Done_{i,j})$ is considered as one of the preconditions of each action $Ac_{i,j}$. Putting it all together, the following forms the template for action $Ac_{i,j}$ encoding event $e_{i,j}$:

$(:$ ACTION $Ac_{i,j}$

: PRECONDITION $(\text{AND } (Done_{i,(j-1)}) \ (\text{NOT } Done_{i,j}) \ldots)$

: EFFECT $(\text{AND } (Done_{i,j}) \ldots)$

$)$

Note that if an actions is encoding the first event in a thread T_i, then its precondition only consists of $(\text{NOT } Done_{i,1})$. The actions may also have other preconditions and effects according to the type of the event (i.e., read, write, thread creation, lock acquire, and release) they represent. The ... denotes that other event-specific conditions may be added to the preconditions and effects of the template. In the following, we describe each of these event types in detail.

Write Events: Let W_x represent the set of all write events to variable x in the observed run. To keep track of the most recent write event to variable x, we consider a set of predicates, represented by $writes(x) = \{(x_{m,n}) |\ e_{m,n} \in W_x\} \bigcup \{(x_{init})\}$.

Predicate (x_{init}) indicates whether x has its initial value. It is initially true, indicating that no write event has been performed to x. Predicate $(x_{m,n})$ indicates whether event $e_{m,n}$ has performed the most recent write to x. Predicates of this type are all initially false. When action $Ac_{m,n}$, which encodes a write to x, is applied, predicates $(x_{m,n})$ becomes true. In addition, all predicates in $writes(x)$ other than $(x_{m,n})$, are set to false indicating that they are not the most recent write to x. Therefore, at each point in time only one of the predicates in $writes(x)$ can be true. The following shows how action $Ac_{i,j}$, which corresponds to a write event to variable x, is encoded:

(: ACTION $Ac_{i,j}$

: PRECONDITION (AND $(Done_{i,(j-1)})$ (NOT $Done_{i,j}$))

: EFFECT(AND $(Done_{i,j})$ $(x_{i,j})\forall p \in [writes(x)\backslash\{(x_{i,j})\}]$: (NOT (p)))

)

Read Events: According to data-validity in the maximal causal model, each read event in the predicted run should read the same value as it did in the original run. However, we are not encoding the real values in the planning domain and it is just enough to recognize for each read event, the set of write events that write the same value to the corresponding variable as the read event read in the observed run.

Suppose that event $e_{i,j} = (T_i, rd(x, val))$ reads the value val from variable x and let $Write_{x,val}$ denote the set of events that write value val to variable x. In any precisely predictable run, read event $e_{i,j}$ can only be coupled with a write event in $Write_{x,val}$. Therefore, for each write event $e_{m,n} \in Write_{x,val}$ an action is considered as follows which forces read event $e_{i,j}$ to be coupled with write event $e_{m,n}$:

(: ACTION $Ac_{i,j}\_coupled_{m,n}$

: PRECONDITION (AND $(Done_{i,(j-1)})$ (NOT $Done_{i,j}$) $(x_{m,n})$)

: EFFECT (AND $(Done_{i,j})$)

)

Having $(x_{m,n})$ as the precondition of the action would force the read event to read the value provided by the write event $e_{m,n}$.

Thread Creation Events: According to creation-validity in the maximal causal model, each thread can start execution only after it is created. For each thread T_i, we consider a predicate $(Created_i)$ which indicated whether T_i has been created or not. Each of these predicates $(Created_i)$ is initially false and is set to true by the action corresponding to the event creating T_i. Furthermore, $Ac_{i,1}$ requires $(Created_i)$ to be true in its preconditions.

Lock Acquire and Lock Release Events: According to lock-validity, each lock can be obtained by at most one thread at each point of time. Therefore, if a lock is obtained by thread T_i

then other threads cannot acquire it unless T_i releases the lock. Assume that $L = \{l_1, ..., l_m\}$ is the set of locks acquired in the observed run. To guarantee lock-validity, a predicate $(Available_{l_j})$ is introduced for each lock l_j, indicating whether lock l_j is obtained by any thread or is free. These predicates are initially true since all of the locks are available at the beginning.

Suppose that action $Ac_{i,j}$ corresponds to a lock acquire event on lock l. It requires $(Available_l)$ to be true as a precondition and sets $(Available_l)$ to false as an effect of the action.

(: ACTION $Ac_{i,j}$

: PRECONDITION (AND $(Done_{i,(j-1)})$ (NOT $Done_{i,j}$) $(Available_l)$)

: EFFECT (AND $(Done_{i,j})$ (NOT $(Available_l)$))

)

Having $(Available_l)$ as a precondition requires lock l to be available before the application of the action. Note that after performing the action, lock l is not available anymore and cannot be acquired by any other thread.

Actions corresponding to a lock release event of lock l set $(Available_l)$ to true, making the lock available again. Therefore, action $Ac_{i,j}$ which corresponds to a lock release event on lock l is encoded as follows:

(: ACTION $Ac_{i,j}$

: PRECONDITION (AND $(Done_{i,(j-1)})$ (NOT $Done_{i,j}$))

: EFFECT (AND $(Done_{i,j})$ $(Available_l)$)

)

A domain description following these template transformations of a given trace will enable us to generate feasible runs based on the maximal causal model. Next, we will show how we can predict feasible runs that realize a *null-WR* pair.

Predicting Runs for a *null-WR* Pair

A *null-WR* pair $\alpha = (e, f)$ consists of a write e, writing a null value to a shared variable, and a read f from the same shared variable, reading a non-null value. A null read with respect to α

happens when f reads the null value written by e.

Suppose that $Ac_{i,j}$ is the action corresponding to the write event e which writes a null value to variable x, and Ac_f is the action corresponding to the read event f. Recall that after the application of action $Ac_{i,j}$, predicate $(x_{i,j})$ becomes true indicating that $Ac_{i,j}$ has performed the most recent write to x. According to the encoding of write events, any other write to variable x after the application of $Ac_{i,j}$ would set $(x_{i,j})$ to false. Since e should be the most recent write to x when the read f is about to happen, it is enough to make $(x_{i,j})$ a precondition of action Ac_f.

We also consider a predicate $(Happened_f)$ which represents whether action Ac_f has happened or not. This predicate is initially false and is set to true in the effect set of action Ac_f. Then, the final-state goal is defined as $(: goal(Happened_f))$. In this case, f is guaranteed to read the null value written by e when $(Happened_f)$ becomes true because Ac_f can be applied only when e is the most recent write to x.

We present a simple example of a null-pointer dereference prediction problem in Appendix A and provide the planning encoding of the problem based on the encoding presented in this section.

2.6 Evaluation

We implemented a tool, named EXCEPTIONULL, for predicting null-pointer dereferences in multi-threaded Java programs. EXCEPTIONULL is built on top of PENELOPE [82] which is a tool that predicts atomicity violations using a lock-based analysis. The implementation is equipped with both logical constraint and planning encoding techniques. To evaluate our null-pointer dereference prediction technique, we subjected EXCEPTIONULL to a benchmark suite of multi-threaded Java programs. In the following, we briefly discuss EXCEPTIONULL and then present our experimental results.

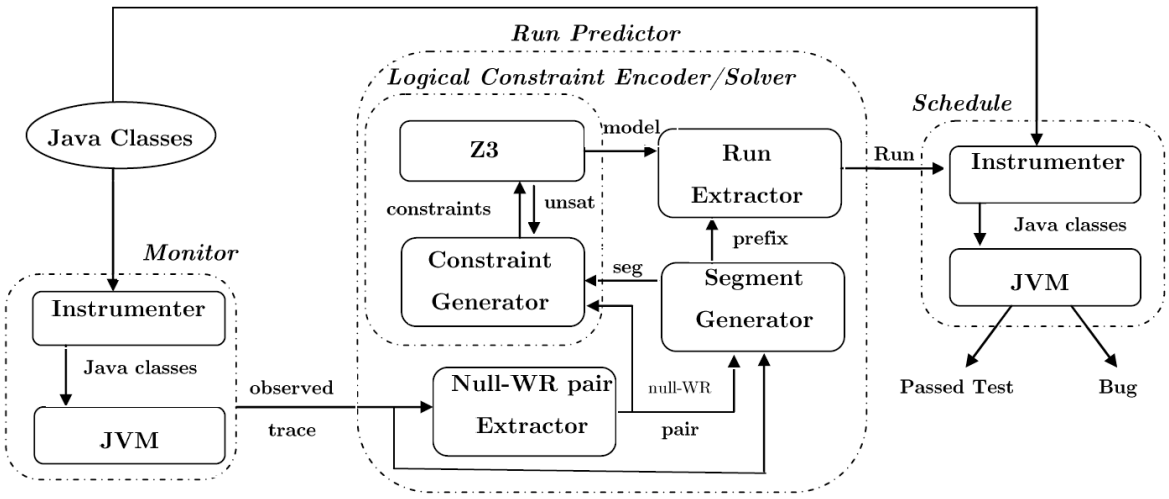


Figure 2.5: EXCEPTIONULL with logical constraint encoder/solver built on top of PENELOPE [82] framework.

2.6.1 Implementation

Figure 2.5 demonstrates the architecture of EXCEPTIONULL with logical constraint encoder/solver. It consists of three main components: a monitor, a run predictor, and a scheduler. The monitor and scheduler are built on top of PENELOPE, with considerable enhancements and optimizations, including the extension of the monitoring to observe values of shared variables at reads and writes. In the following, we will explain each of these components in more detail.

Monitor: The monitor component has an instrumenter which uses the Bytecode Engineering Library (BCEL)[2] to (automatically) instrument every class file in bytecode so that a *call* to an event recorder is made after each *relevant* action is performed. These relevant actions include field and array accesses, acquisitions and releases of locks, thread creations, but exclude accesses to local variables. The instrumented classes are then used in the Java Virtual Machine (JVM) to execute the program and generate a global trace. For the purpose of generating data-validity constraints, values read/written by shared variable accesses are also recorded.

Run Predictor: The run predictor consists of several components: *null-WR* pair extractor,

[2]http://jakarta.apache.org/bcel

segment generator, logical constraint and planning encoder/solver, and run extractor. The *null-WR* pair extractor generates a set of *null-WR* pairs from the observed trace by the static lock analysis described in Section 2.3.1. The segment generator component, for each *null-WR* pair $\alpha = (e, f)$, isolates a part of the observed trace π that is relevant to α as described in Section 2.3.2 and passes it to the logical constraint or planning encoder/solver.

Given a *null-WR* pair and the relevant segment, the logical constraint encoder/solver (as shown in Figure 2.5), first, produces a set of constraints according to the encoding presented in Section 2.4. Then, it utilizes the Z3 [9] SMT solver to find a solution. Any model found by Z3 represents a partial run. The run extractor component generates a run by attaching the partial run generated based on the model returned by Z3 to the prefix generated by the segment generator. When Z3 cannot find a solution, the logical constraint encoder/solver iteratively weakens the constraints according to the relaxation method proposed in Section 2.4.2 and calls Z3 until a solution is found or a threshold is reached.

Figure 2.6 shows the planning encoder/solver. Given a *null-WR* pair and the relevant segment, the planning encoder/solver, first, uses the segment to generate a planning problem encoding the maximal causal model as discussed in Section 2.5.2. Then, *null-WR* pair is encoded as a temporally extended goal. The planning domain and the temporally extended goal are then compiled into a classical planning problem according to the algorithm proposed in Section 2.5.2. To find a plan, one can use a variety of plan generation algorithms. Here we use FF, which is a domain independent planning system. FF is complete in that if a planning goal is reachable, FF is guaranteed to find it.

Scheduler: The scheduler is implemented using BCEL as well; the scheduling algorithm is instrumented into Java classes using bytecode transformations, so that the program interacts with the scheduler when it is executing an action regarding shared variable accesses or synchronization events. The scheduler, at each point, looks at the predicted run, and directs the appropriate thread to perform a sequence of n steps. The communication between the scheduler and threads is implemented using wait-notify synchronization which allows us to have a

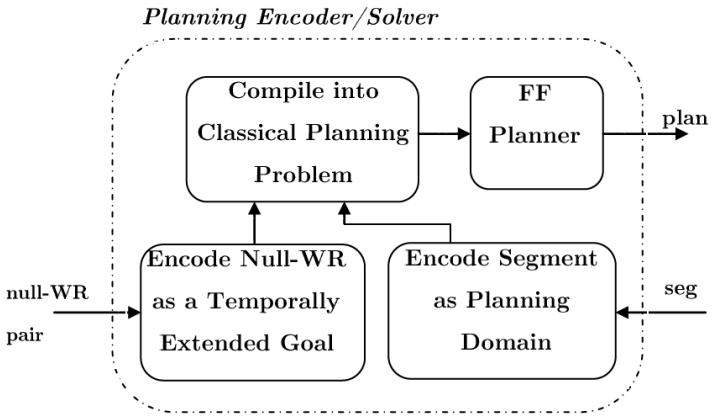


Figure 2.6: Planning encoder/solver component in run prediction.

finely orchestrated scheduling process.

2.6.2 Experiments

We evaluate the effectiveness and efficiency of our null-pointer dereference prediction technique by subjecting EXCEPTIONULL to a benchmark suite of 13 concurrent programs, against several test cases and input parameters. We investigate the effects of the static pruning and relaxed prediction discussed in Sections 2.3.2 and 2.4.2, respectively. Finally, we evaluate utilizing planning techniques in null-pointer dereference prediction (as discussed in Section 2.5).

Benchmarks: The benchmarks are all concurrent Java programs that use `synchronized` blocks and methods as means of synchronization. They include `RayTracer` from the Java Grande multi-threaded benchmarks[3], `elevator` from ETH [88], `StringBuffer`, `Vector`, `Stack`, and `HashSet` from Java libraries, `Pool` (3 releases) and `StaticBucketMap` from the Apache Commons Project[4], `Apache FtpServer`[5], `Hedc`[6], and `Weblech`[7]. The `elevator` program simulates multiple lifts in a building, `RayTracer` renders a frame of an arrangement of spheres from a given view point, `Pool` is an object pooling API in the Apache Commons

[3]http://www.javagrande.org/
[4]http://commons.apache.org
[5]http://mina.apache.org/ftpserver
[6]http://www.hedc.ethz.ch
[7]http://weblech.sourceforge.net

			Monitoring					Prediction			Scheduling				
Program (LOC)	Input	Base	Num. of Threads	Num. of Shared Variables	Num. of Locks	Num. of Potential Interleaving Points	Time to Monitor	Num. of *null-WR* Pairs	Num. of Precisely Predicted Runs	Additional Predicted Runs by Relaxation	Num. of Feasible Predictions	Average Time per Predicted Run	Total Time	Null Pointer Deref. by Precise Prediction	Additional Null-Pointer Deref. by Relaxation
Elevator (566)	Data	7.3s	3	116	8	14K	7.4s	0	-	-	-	-	7.9s	**0**	**0**
	Data2	7.3s	5	168	8	30K	7.4s	0	-	-	-	-	8.9s	**0**	**0**
	Data3	19.2s	5	723	50	150K	19.0s	0	-	-	-	-	58.5s	**0**	**0**
RayTracer (1.5K)	A-10	5.0s	10	106	10	648	5.0s	9	9	-	9	5.6s	50.5s	**1***	**0**
	A-20	3.6s	20	196	20	1.7K	4.4s	19	19	-	19	6.7s	2m15s	**1***	**0**
	B-10	42.4s	10	106	10	648	42.5s	9	9	-	9	42.7s	6m24s	**1***	**0**
Pool 1.2 (5.8K)	PT1	<1s	4	28	1	98	<1s	3	2	1	3	<1s	1.6s	**2**	**0**
	PT2	<1s	4	29	1	267	<1s	3	0	0	-	-	8.8s	**0**	**0**
	PT3	<1s	4	20	3	180	<1s	26	0	23	16	1.2s	27.0s	**0**	**3**
	PT4	<1s	4	24	3	360	<1s	32	2	21	15	2.5s	57.8s	**0**	**1**
Pool 1.3 (7K)	PT1	<1s	4	30	1	100	<1s	3	0	3	3	<1s	2.6s	**0**	**0**
	PT2	<1s	4	31	1	271	<1s	3	0	0	-	-	9.8s	**0**	**0**
	PT3	<1s	4	20	3	204	<1s	35	0	30	19	1.4s	42.9s	**0**	**0**
	PT4	<1s	4	23	3	422	<1s	62	1	48	29	2.2s	1m49s	**0**	**1**
Pool 1.5 (7.2K)	PT1	<1s	4	33	2	124	<1s	2	0	1	1	1.5s	1.5s	**0**	**0**
	PT2	<1s	4	34	2	306	<1s	5	0	1	0	10.5s	10.5s	**0**	**0**
	PT3	<1s	4	15	2	108	<1s	3	0	0	-	-	4.1s	**0**	**0**
	PT4	<1s	4	18	2	242	<1s	18	1	7	8	3.4s	27.4s	**0**	**1**
SBucketMap (750)	SMT	<1s	4	123	19	892	<1s	2	2	-	2	<1s	1.3s	**1**	**0**
Vector (1.3K)	VT1	<1s	4	44	2	370	<1s	21	11	10	21	<1s	14.3s	**2**	**0**
	VT2	<1s	4	34	2	536	<1s	31	21	10	31	1.1s	33.0s	**1**	**0**
	VT3	<1s	4	34	2	443	<1s	32	22	10	32	<1s	22.1s	**1**	**0**
	VT4	<1s	4	29	2	517	<1s	30	0	30	30	2s	59.4s	**0**	**1***
	VT5	<1s	4	29	2	505	<1s	85	1	84	82	2s	2m57s	**0**	**1***
Stack (1.4K)	ST1	<1s	4	29	2	205	<1s	11	6	5	11	<1s	5.5s	**2**	**0**
	ST2	<1s	4	24	2	251	<1s	16	11	5	15	<1s	10.9s	**1**	**0**
	ST3	<1s	4	24	2	248	<1s	17	12	5	17	<1s	10.3s	**1**	**0**
	ST4	<1s	4	29	2	515	<1s	30	0	30	30	1.8s	53.2s	**0**	**1***
	ST5	<1s	4	29	2	509	<1s	85	1	84	83	2.0s	2m51s	**0**	**1***
HashSet (1.3K)	HT1	<1s	4	76	1	432	<1s	7	7	-	7	<1s	3.2s	**1**	**0**
	HT2	<1s	4	54	1	295	<1s	0	-	-	-	-	<1s	**0**	**0**
StringBuffer (1.4K)	SBT	<1s	3	16	3	80	<1s	2	2	-	2	<1s	1.3s	**1**$^{+}$	**0**
Apache FtpServer (22K)	LGN	1m2s	4	112	4	582	60s	116	78	32	65	1m13s	2h14m46s	**9**	**3**
Hedc (30K)	Std	1.7s	7	110	6	602	1.74s	18	9	1	10	11.7s	1m57s	**1**	**0**
Weblech v.0.0.3 (35K)	Std	4.9s	3	153	3	1.6K	4.92s	55	10	29	30	16.26s	10m34s	**1**	**1**$^{@}$
											Total Number of Errors			**27**	**14**

Table 2.1: Experimental results for precise/relaxed prediction using logical constraint encoder/-solver. Symbols *, $^{+}$, and $^{@}$ represent test harness failure, array-out-of-bound exception, and unexpected behavior, respectively. All other errors are null-pointer dereference exceptions.

suite, `StaticBucketMap` is a thread-safe implementation of the Java Map Interface, `Apache FtpServer` is a FTP server by Apache, and `Vector`, `Stack`, `HashSet` and `StringBuffer` are Java libraries that respectively implement a concurrent vector, stack, HashSet and StringBuffer data structures. `Hedc` is a Web crawler application and `Weblech` is a websites download tool.

Table 2.1 illustrates the experimental results for null-pointer dereference prediction using the logical constraint encoder/solver. It provides information about monitoring, run prediction, and scheduling phases. In the monitoring phase, the number of threads, shared variables, locks, the number of potential interleaving points (i.e., number of global events), and the time taken for monitoring are reported. For the prediction phase, we report the number of *null-WR* pairs in the observed run, the number of precisely predicted runs, and the additional number of runs predicted by relaxation (when there is no precisely predicted run for a null read-write pair). In the scheduling phase, we report the total number of feasible (i.e., could be scheduled) predicted runs. Finally, we report the average time for prediction and rescheduling of each run, the total time taken to complete the tests (for all phases), and also the number of errors found using the precise and relaxed prediction.

Observations: Comparing the number of *null-WR* pairs with the number of precisely predicted runs, we can see that our precise prediction technique performs very well in practice; i.e., for most of the benchmarks, the precise prediction is able to generate feasible runs for many of the *null-WR* pairs . Furthermore, we could find 27 errors in total in our set of benchmarks using the precise prediction technique that also proves the effectiveness of the precise prediction.

According to the number of additional runs predicted by relaxation, we conclude that relaxation technique works extremely well; The relaxation method could predict lots of runs for the *null-WR* pairs for which the precise prediction technique cannot find any solution. Comparing the number of feasible predictions and number of precisely predicted runs, we can see that a large number of runs predicted by the relaxed prediction technique were feasible. Furthermore, we could find 14 additional errors by the relaxed prediction technique.

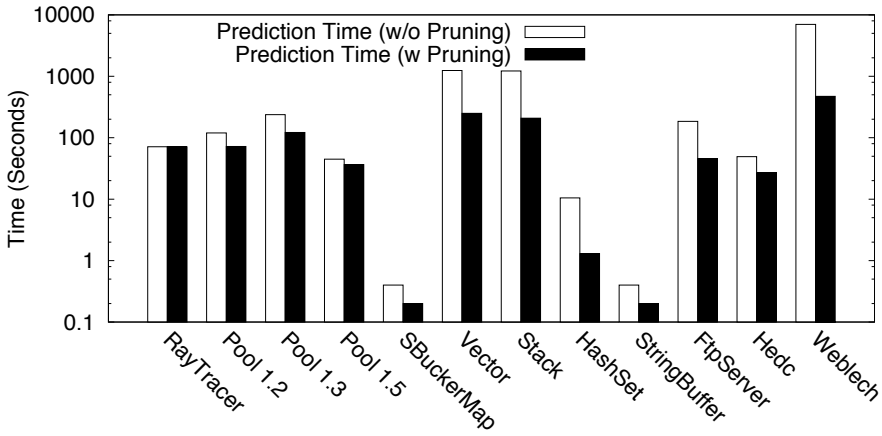


Figure 2.7: Prediction times with/without pruning in log scale.

In our experiments, the errors manifested in the form of raised exceptions in most of the programs. In `Weblech`, in addition to a null-pointer dereference, an unwanted behavior occurred (the user is asked to push a stop button even after the website is downloaded completely, resulting in non-termination!). `RayTracer` has a built-in validation test which was failed in some of the predicted runs. For some of the test cases of `Vector` and `Stack` the output produced was not the one expected. In Table 2.1, exceptions raised in different parts of the code are counted as separate errors. For example, the 9 exceptions in `FtpServer` are raised in 7 different functions and at different locations inside the functions, and involve null-pointer dereferences on 5 different variables.

In general, we can see that EXCEPTIONULL performs considerably well, predicting a large number of feasible program runs leading to null-pointer dereferences. In total, it finds about 40 executions with null-pointer dereferences in the benchmarks. All the errors are completely reproducible deterministically using the scheduler. Furthermore, despite the use of fairly sophisticated static analysis and logic-solvers, the time taken for prediction is very reasonable.

The effect of pruning: Figure 2.7 illustrates the substantial impact of our pruning algorithm (presented in Section 2.3.2) in reducing prediction time. It presents prediction time with and without using the pruning algorithm. Note that the histogram is on a logarithmic scale. For ex-

ample, in the case of `Weblech`, the prediction algorithm is about 16 times faster with pruning. Furthermore, all errors found without the pruning were found on the pruned runs, showing that the pruning did not affect the quality of error-finding on our benchmarks.

Effectiveness and efficiency of planning encoder/solver: To evaluate the planning encoding of predicting null-pointer dereferences (presented in Section 2.5.2), we apply EXCEPTIONULL using the planning encoder/solver on some of the benchmarks. As mentioned before, the encoding is based on the maximal causal model and can only support precise prediction. Therefore, from the Java benchmarks, we pick ones for which relaxed prediction does not have any effect regarding the number of errors found. In Table 2.2, in addition to the information about the observed runs, i.e., number of threads, shared variables, locks and events, we provide the number of *null-WR* pairs, number of precisely predicted runs, average time per prediction and number of null-pointer dereferences found by the predicted runs for both logical constraint and planning encodings.

Although we have used FF (which is a heuristic-based planner) in our experiments, one can observe that the number of predicted runs is not affected by the heuristics used in FF. This is because FF is complete in that if a planning goal is reachable, FF is guaranteed to find it. Therefore, according to the number of predicted runs logical constraint encoder/solver and planning encoder/solver are equal to each other. As a result, using the planning encoder/solver we could find all of the errors found by using the logical constraint encoder/solver; in fact, the errors found in both approaches are the same.

Another observation is that the planning encoder/solver is much more faster than the logical constraint encoder/solver. The average time per prediction is 0.01 second for FF which is negligible. This implies that the heuristic-based search algorithm embedded in FF can perform very well on the planning problems obtained through the encoding outlined in Section 2.5.2. This suggests that maybe other test generation techniques also can employ a planning encoder/solver to speed up and benefit from the advance search algorithms embedded in planners.

As noted previously, the novelty and effectiveness of the planning encoding is in the con-

		Run Information				Prediction						
Program (LOC)	Input	Num. of Threads	Num. of Shared Vars	Num. of Locks	Num. of Potential Interleaving Points	Num. of *null-WR* Pairs	Num. of Precisely Predicted Runs by FF	Num. of Precisely Predicted Runs by SMT	Average Time per Predicted Run by FF (s)	Average Time per Predicted Run by SMT (s)	Null Pointer Deref. by Precise Prediction by FF	Null Pointer Deref. by Precise Prediction by SMT
`Vector` (1.3K)	VT1	4	44	2	370	21	11	11	0.01	0.33	2	2
	VT2	4	34	2	536	31	21	21	0.01	0.39	1	1
	VT3	4	34	2	443	32	22	22	0.01	0.34	1	1
`Stack` (1.4K)	ST1	4	29	2	205	11	6	6	0.01	0.63	2	2
	ST2	4	24	2	251	16	11	11	0.01	0.75	1	1
	ST3	4	24	2	248	17	12	12	0.01	0.65	1	1
`SBucketMap` (750)	BMT	4	123	19	892	2	2	2	0.01	0.25	1	1
`Pool 1.2` (5.8K)	PT1	4	28	1	98	3	2	2	<0.01	0.27	2	2
	PT2	4	29	1	267	0	0	0	-	-	-	-
	PT4	4	24	3	360	32	2	2	<0.01	1.5	0	0
`Pool 1.3` (7K)	PT1	4	30	1	100	3	0	0	-	-	-	-
	PT2	4	31	1	271	3	0	0	-	-	-	-
	PT4	4	23	3	422	62	1	1	<0.01	1.33	0	0
`HashSet` (1.3K)	HS1	4	76	1	432	7	7	7	0.01	0.19	1	1
	HS2	4	65	1	295	0	0	0	-	-	-	-
`StringBuffer` (1.4K)	SBT	3	16	3	80	2	2	2	<0.01	0.15	1	1
`Elevator` (566)	Data	3	116	8	14K	0	-	-	-	-	-	-

Table 2.2: Experimental results for precise prediction using planning encoder/solver.

		Data Races			Atomicity Violations		
Program (LOC)	Input	Num. of Access Patterns	Num. of Predicted Runs	Avg. Time per Predicted Run by FF (s)	Num. of Access Patterns	Num. of Predicted Runs	Avg.Time per Predicted Run by FF (s)
Vector (1.3K)	VT1	0	-	-	0	-	-
	VT2	0	-	-	11	11	0.01
	VT3	0	-	-	4	4	0.01
Stack (1.4K)	ST1	0	-	-	0	-	-
	ST2	0	-	-	22	22	0.01
	ST3	0	-	-	8	8	0.01
SBucketMap (750)	BMT	1	1	0.02	2	2	0.01
Pool 1.2 (5.8K)	PT1	2	2	<0.01	1	1	<0.01
	PT2	7	2	<0.01	15	10	<0.01
	PT4	7	1	<0.01	57	18	<0.01
Pool 1.3 (7K)	PT1	0	-	-	0	-	-
	PT2	0	-	-	12	10	<0.01
	PT4	0	-	-	127	10	<0.01
HashSet (1.3K)	HT1	20	20	0.01	3	3	0.01
StringBuffer (1.4K)	SBT	0	-	-	1	1	<0.01
Elevator (566)	Data	0	-	-	4	0	-

Table 2.3: Experimental results for predicting data races and atomicity violations using planning encoder/solver.

ceptualization and realization of the test generation task as an AI automated planning task where the null reads patterns are characterized as temporally extended goals. Nevertheless, our conceptualization of the test generation task allows for predicting runs with other violation patterns as well. This ability to elegantly and effectively deal with arbitrary violation patterns is a strength of the approach. We investigated the applicability of this test generation technique for predicting data races and atomicity violations as well.

We used PENELOPE [82] to extract a set of data race and atomicity violation patterns from the observed runs. Each data race pattern is a tuple (e, f) where e and f are accesses to the same shared variable in different threads which conflict with each other (i.e., least one of them is a write access). Each atomicity violation pattern is a tuple (e_1, e_2, f) where e_1, e_2 and f are accesses to the same shared variable, e_1 and e_2 are in one thread and f is in another thread, and f conflicts with both e_1 and e_2. For each data race pattern (e, f), the planning goal is to perform the actions corresponding to e and f, back to back. For each atomicity violation pattern (e_1, e_2, f) the planning goal is to perform the corresponding action of f after the corresponding action of e_1 (and before the corresponding action of e_2) is performed.

Table 2.3 shows the experimental results for predicting data races and atomicity violations using planning encoder/solver. We report the number of access patterns, number of predicted runs, and average time per predicted run by FF for both data races and atomicity violations. According to the experiments, the planning encoder/solver was able to predict runs for most of the access patterns pretty fast (i.e., average 0.01 sec.). This confirms the applicability of the planning techniques in predicting runs with arbitrary violation patterns.

Conclusion: Our experiments showed that our prediction technique in predicting null-pointer dereferences is very effective in practice. Using this technique, we could find 41 bugs in our set of benchmarks. We showed that our relaxation technique is very useful when there is no precise solution for the prediction problem. According to the relaxation technique, we could predict runs (actually many of them were feasible runs) that resulted in finding 13 additional bugs. We also showed that our pruning technique made the prediction process up to 16 times

faster for some of the benchmarks without affecting the quality of bug-finding. We compared the logical constraints and AI planning encoding approaches for the precise prediction. Our experiments showed that the planning encoder/solver is much more faster than the logical constraints encoder/solver. Finally, we showed that our prediction technique is general and can be applied in predicting runs with other violation patterns.

2.7 Related Work

Prediction techniques use heuristics (e.g., atomicity violations, data races, and assertion violations) to reduce the interleaving exploration space under fixed inputs. Similar to our prediction technique for finding null-pointer dereferences, several prediction techniques work at shared communication level, i.e., accesses to shared variables and synchronization events. These techniques either enhance a lock-based analysis [17, 82, 77, 78, 35] or a graph-based analysis [93, 92] for prediction. For example, PENELOPE [17, 82] is a testing tool for predicting atomicity violations in concurrent program. It works at shared communication level and uses a lock-based analysis that guarantees all of the predicted runs respect the semantics of locking. However, since data-flow is ignored in the analysis, the predicted runs are not guaranteed to be feasible. Other lock-based techniques [77, 78, 35] follow a more restrictive approach in interleaving exploration to guarantee soundness; in the predicted runs each read should be matched with exactly the *same write* as it was matched in the observed run. Techniques that utilize a graph-based analysis [93, 92], on the other hand, build a dependency graph based on the events in the observed run to identify atomicity violations. These techniques are even more restrictive than lock-based techniques according to the set of explored interleavings since in addition to the constraint that each read should be matched with exactly the same write as it did in the observed run, the predicted runs should preserve the order of lock blocks as in the observed run. These restrictions guarantee soundness for the prediction technique. The maximal causal model (MCM) [74] is another technique working at the shared communication level that targets

sound prediction. It is the maximal precise prediction technique one can achieve at the shared communication level. We discussed this technique in this chapter in detail. It has been used by Said et al. [68] for finding data race witnesses. Our null-pointer prediction technique utilizes MCM to guarantee soundness.

There are also prediction techniques [90, 89] that works at the *statement* level. These techniques observe a run and symbolically encode every single instruction executed (local computation as well as global computation) in the observed run and use a sound symbolic analysis to predict atomicity violations and assertion violations. These techniques have too big an overhead to scale to large executions.

A more liberal notion of generalized dynamic analysis of a single run has also been studied in a series of papers by Chen et al. [7, 6]. JPREDICTOR [7] offers a predictive runtime analysis that uses *sliced causality* [6] to exclude the irrelevant causal dependencies from an observed run and then exhaustively investigates all of the interleavings consistent with the sliced causality to detect potential errors.

CTRIGGER [59] is another testing tool that targets finding atomicity violations in concurrent programs. It first extracts a set of atomicity violation patterns from an observed run. Then, for each pattern, it instruments the program code by inserting some synchronization around the accesses corresponding to the pattern, with the aim of increasing the probability of realizing the violation pattern in the execution of the instrumented program. However, the atomicity violation patterns are not guaranteed to be realized in the instrumented program, i.e., it is not sound.

Another closely related work is CONMEM [96], where the authors target a variety of memory errors in testing concurrent programs, including null-pointer dereferences, but the prediction algorithms are much weaker and quite inaccurate compared to our robust prediction techniques; their prediction analysis is mostly based on the synchronization events present in an observed run (ignoring the flow of data among the threads) and hence is not sound. Therefore, they had to build a validator to automatically prune false positives by enforcing the predicted

interleavings.

2.8 Summary

In this chapter, we introduced a new pattern for interleaving selection that targets null-pointer dereferences in concurrent programs. We utilized a carefully chosen set of techniques for sound and scalable prediction. For the sake of scalability, our prediction is based on an approximation that ignores local computation entirely. We also proposed a static pruning technique that reduces the size of the prediction problem drastically. We exploited the maximal causal model [74] that guarantees sound prediction at shared communication level. For cases where there is no sound solution, we proposed a relaxation method at the expense of losing soundness guarantees. However, our experiments showed that the majority of the runs predicted by the relaxation method are feasible. We developed two different encodings for our prediction problem based on logical constraints and AI planning. The former encoding allows us to use state-of-the-art SMT solvers to search for a solution. The latter encoding allows us to benefit from the compact encoding and advanced heuristic-based search algorithms embedded in the planners. According to our experiments, both approaches are equal regarding the effectiveness in bug finding. However, the planning approach showed to be much more faster than the logical constraints approach. We implemented our prediction technique in a tool that predicts null-pointer dereferences in Java multi-threaded programs. We performed some experiments, based on our tool, that proved the efficiency and the effectiveness of our null-pointer dereference prediction technique.

Chapter 3

Test Generation Based on Under-approximations of Programs

Program slices built from concurrent program executions, referred to as *concurrent trace programs*, have been used as program under-approximations to find bugs in the corresponding programs [89, 77, 80, 79]. Concurrent trace programs encode program runs as a set of thread-local computations and a set of inter-thread communications on shared variables or synchronization operations. Some techniques [80, 79] subject concurrent trace programs (instead of the *whole* program) to input/interleaving exploration for finding assertion violations. However, none of these techniques target test generation (i.e., input/schedule generation) for exploring different program behaviours. Rather, they focus on finding assertion violations corresponding to assertions that *present* in the approximation model. Furthermore, in all of these techniques, the approximation model is fixed at the beginning, and none of these techniques consider program behaviours that are beyond the approximation model.

Here, we also utilize concurrent trace programs as under-approximations of concurrent programs. However, our main goals are (i) to use the approximation model as a basis for test generation, and (ii) to generate tests that increase code coverage in the concurrent program. More specifically, test generation targets covering *static* branches in the program that have not

been covered by previous tests. Note that in an *active testing framework* [25], many runtime bugs can be encoded as branches. Therefore, by targeting branch coverage, we can implicitly aim for catching those bugs.

We develop a heuristic, based on exploiting *interesting def-use pairs*, where a definition (def) represents a write to a shared variable in some thread and a use represents a read from that variable in some other thread, that would lead to covering a previously uncovered branch. We exploit the fact that it is easy to generate a set of different test runs (e.g., by executing the program with different input values) without any significant effort. By observing already available test runs for various writes to shared variables, we are then able to select segments of previously observed runs, and insert them (not necessarily atomically) into other runs to exploit previously unseen def-use pairs leading to covering previously uncovered branches of the program. In the following, we call such segments containing a write to a shared variable *interloper* segments.

Given this intuition, we have to address the following challenges: (1) How to generate an interesting set of test inputs and thread schedules to start with, if none is provided, (2) How to effectively search for *feasible* interloper segments, and (3) How to generate inputs and feasible schedules corresponding to inserting an interloper segment into another run leading to covering a previously uncovered branch.

To address the first question, we rely on sequential test generation techniques; i.e., we subject each thread to sequential testing individually, first. We utilize the fact that state-of-the-art sequential test generation techniques are generally able to quickly cover a large part of the program in terms of branches in individual threads. Indeed, the branches of a concurrent program that are not covered using sequential testing techniques alone may require *interesting* interactions between the threads of the concurrent program that are worth further exploration. To address the second question, we develop a static Multi-Trace Analysis (MTA) technique. In our MTA technique, we advance the symbolic predictive analysis technique [89, 77] (which considers fixed inputs) with symbolic inputs to be able to generate input values. Furthermore,

```
1 public void Thread1(){
2   x = 1;
3   if (x > 1)
4       error();
5 }
```

```
6 public void Thread2(int input){
7   x = 0;
8   if(input > 0)
9       x = input;
10  y = x;
11 }
```

Figure 3.1: A concurrent program with a reachable error state.

unlike symbolic predictive analysis, our MTA exploits information in *multiple* program runs. Finally, to address the third question, we generate an appropriate logical constraint system, whose model implies a set of input values and a schedule, and use SMT solvers to search for solutions.

This chapter is based on our publication on MTA technique (i.e., [66]). We elaborate the MTA technique in detail and evaluate it experimentally.

3.1 Motivating Example

Consider the simple concurrent program consisting of two threads that call `Thread1` and `Thread2` in Figure 3.1, respectively. Variable `x` is shared among the threads and `input` is the input of the program. The error in `Thread1` is not reachable (i.e., the `if`-branch is not coverable) when the threads are executed sequentially back to back. However, the error will become reachable when `input` ≥ 2 and the read of `x` at line 3 in `Thread1` reads the value written by the write in other thread at line 9. Our goal is to generate a test (i.e., input values plus a schedule) such that the execution of the program with the generated inputs according to the schedule gets to the error state.

Suppose that we subject each thread to sequential test generation to increase code coverage as much as possible in individual threads. Sequential test generation, will not do much for `Thread1` since its behavior does not depend on the inputs of the program. However, for

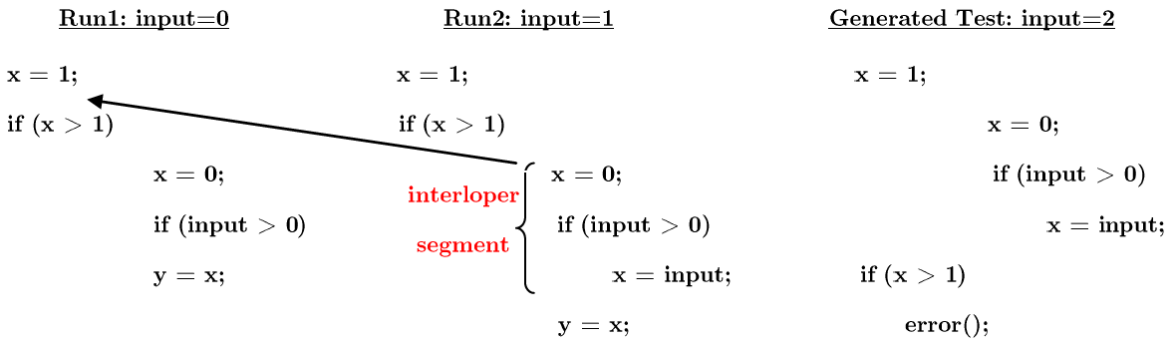


Figure 3.2: Test generation based on MTA for the program in Figure 3.1.

`Thread2`, sequential test generation will generate two different values for `input`, one ≤ 0 and one > 0, corresponding to skipping and covering the `if`-branch at line 8, respectively. Without loss of generality, suppose that 0 and 1 are the two values generated for `input`.

Now, we execute the concurrent program with the generated inputs and get two concurrent runs `Run1` and `Run2` depicted in Figure 3.2. One can see that the `if`-branch in `Thread1` (which leads to the error) is skipped in both of these runs since the read of `x` in the branch condition always reads the value written to `x` locally at line 2. However, we observe that in `Run2` there is a write to `x` by the other thread that could be a candidate for providing a value for the read of `x` in the branch condition (overwriting the value written at line 2). Therefore, we select an appropriate *interloper* segment from `Run2`, containing the candidate write to variable `x` (as shown in Figure 3.2), and insert it between the write to `x` at line 2 and the read from `x` at line 3 in `Run1` (shown by an arrow in Figure 3.2) and search for input values and a schedule that would result in covering the `if`-branch at line 8 (if possible). Figure 3.2, on the right side, depicts a generated test that results in covering the `if`-branch at line 8 and hence leads to the error state. Note that the error state cannot be reached by applying prediction techniques (which work on fixed inputs) on `Run1` or `Run2`.

Similar to symbolic prediction [89, 90], we symbolically encode the set of all feasible runs, where the interloper segment is inserted in `Run1` and the `if`-branch at line 3 is taken, as a set of logical constraints. Unlike symbolic prediction which works on fixed inputs, we consider

symbolic values for inputs to be able to do input generation. In this example, the interloper segment is inserted atomically between the write to `x` and the read from `x` in `Run1`. However, in general, these two events might be far from each other and in that case, the generated constraints encode all feasible runs in which the statements in the interloper segment are interleaved with the statements in `Run1` appearing between the corresponding write and read. We use the state-of-the-art SMT solvers to find a solution.

3.2 Preliminaries

In this section, we first formally define symbolic traces and concurrent trace programs which form the basis for our test generation technique. Then, we discuss how a concurrent trace program, obtained from a single execution of a concurrent program, is used by symbolic prediction techniques to predict bugs in the concurrent program. Finally, we provide a brief overview of concolic testing for sequential programs.

3.2.1 Symbolic Traces

In Section 2.2.1, we defined the notion of a *global trace* as the sequence of events corresponding to accesses to shared variables and synchronization events. In this section, we define the notion of a *symbolic trace* that contains information about both global and local computation in program executions.

For a concurrent program, let T= $\{T_1, T_2, ...\}$ represent the set of thread identifiers, and SV be the set of shared variables. Each thread T_i has a finite set of *local variables* LV_i, and can access the set of variables in $V_i = SV \cup LV_i$ during its execution. Each thread T_i executes a set of *trace statements*.

Definition 3.2.1 (Trace Statement)**.** *A trace statement is a tuple* $(sId, stmt)$ *where* sId *is a unique identifier (e.g., a combination of a thread identifier and location in the program), and* $stmt$ *is one of the following forms:*

- $(\mathsf{assume}(c), asgn)$ *is the atomic* guarded assignment*, where* $asgn$ *is a set of assignments, each of the form* $v := exp$*, where* $v \in V_i$ *is a variable and* exp *is an expression over* V_i*.* $\mathsf{assume}(c)$ *means the conditional expression* c *over* V_i *must be true for the assignments in* $asgn$ *to execute.*

- $\mathsf{assert}(c)$ *is the assertion statement. The conditional expression* c *over* V_i *must be true when the statement is executed; otherwise, an error is raised.*

The guarded assignment $(\mathsf{assume}(c), asgn)$ may have the following variants: (1) when $c = \mathsf{true}$, it can represent normal assignments; (2) when the assignment set is empty, $\mathsf{assume}(c)$ itself can represent the `then`-branch of an `if(c)`-statement, while $\mathsf{assume}(\neg c)$ can represent the `else`-branch; and (3) with both guard and assignments, it can represent an atomic *check-and-set*, which is the foundation for synchronization primitives. In particular, it can precisely capture the semantics of all synchronization primitives in the standard *PThreads* library. For example, acquiring lock lk in thread T_i is modeled as $(\mathsf{assume}(lk = 0), \{lk := T_i\})$.

Let $stmtIds$ represent the set of trace statement identifiers in the program. We refer to the execution of trace statements as *events*. An *event* e is a tuple (tid, loc), where $tid \in T$ is a thread index, $loc = (sId, instId)$ represents the location of thread T_{tid} where $sId \in stmtIds$ is the identifier of the statement and $instId$ represents the thread-local instance identifier of the trace statement with statement identifier sId; i.e., if a trace statement is executed again inside a loop, a new event will be generated at run-time with the same sId and a new $instId$. Let EV denote the set of all possible events.

Definition 3.2.2 (Symbolic Traces)**.** *A symbolic trace of a program is a finite sequence* $\rho \in EV^*$*.*

In this chapter, whenever we refer to traces, we mean symbolic traces.

Definition 3.2.3 (Global Locations in Symbolic Traces)**.** *Let* ρ *be a symbolic trace. The global location at* $\rho[j]$ *is defined as a tuple* $(loc_1, loc_2, \ldots)$ *where* loc_i *is the location of thread* T_i *at* $\rho[j]$*, i.e., the location of* T_i *in the last event of thread* T_i *in* ρ *before* $\rho[j]$*.*

A global location at a specific point in a symbolic trace defines the location of each thread at that point.

3.2.2 Concurrent Trace Program (CTP)

Given a symbolic trace ρ, we build a concurrent program where each thread T_i consists of a single path of execution (specifically the path it took in ρ). We refer to this program as a *concurrent trace program* (or CTP). The semantics of CTPs is defined using state transition systems.

Let $V = \bigcup_{i=1}^{k}\{LV_i\} \cup SV$, be the set of variables and Val be a set of values. A *state* is a map $s : V \to Val$ assigning a value to each variable. We use $s[v]$ and $s[exp]$ to denote variable and expression values in state s, respectively.

A *state transition* $s \xrightarrow{e} s'$, where s, s' are states and $e = (T_i, (sId, instId))$ is an event, exists iff one of these conditions holds:

- sId refers to a statement of form $(\mathsf{assume}(c), asgn)$, $s[c]$ is true, and for each assignment $v := exp$ in $asgn$, $s'[v] = s[exp]$ holds; s and s' agree on other variables. Note that if $s[c]$ is false, the transition does not exist, i.e., the execution is blocked.

- sId refers to a statement of form $\mathsf{assert}(c)$ and $s[c]$ is true. When $s[c]$ is false, an attempt to execute event e raises an error.

Now, we formally define concurrent trace programs obtained from symbolic traces.

Definition 3.2.4. *Let $\rho = e_1, \ldots, e_n$ be a symbolic trace of a concurrent program. A* concurrent trace program *of ρ is a partially ordered set $CTP_\rho = (E, \sqsubseteq)$ such that $E = \{e \mid e \text{ is in } \rho\}$ is a set of events, and $\sqsubseteq$ is a partial order, where for any $e_i, e_j \in E$, we have $e_i \sqsubseteq e_j$ iff $tid(e_i) = tid(e_j)$ and $i < j$ (in ρ, event e_i appears before e_j).*

A concurrent trace program CTP_ρ orders events from the same thread by their execution order in ρ; events from different threads are not *explicitly* ordered. Let $\rho' = e'_1 \ldots e'_n$ be a

linearization of CTP_ρ. ρ' is said to be a *feasible linearization* iff there exist states $s_0, \ldots, s_n$ such that, s_0 is the initial state of the program and for all $i = 1, \ldots, n$ there exists a transition $s_{i-1} \xrightarrow{e'_i} s_i$.

3.2.3 Predicting Bugs Using CTPs

CTPs have been used to predict bugs in concurrent programs [90, 89]: Given a symbolic trace ρ, a model CTP_ρ is derived to *symbolically* check all its feasible linearizations. For this, a logical constraint formula Φ_{CTP_ρ} is created such that Φ_{CTP_ρ} is satisfiable iff there exists a feasible linearization of CTP_ρ. To generate Φ_{CTP_ρ}, CTP_ρ is first transformed into a concurrent static single assignment (CSSA) [45].

CSSA Encoding. The CSSA form has the property that each variable is defined exactly once. A *definition* of variable v is a trace statement that modifies v, and a *use* is a trace statement where v appears in an expression. Unlike in the classic sequential SSA form, we do not need to add ϕ-functions to model the confluence of multiple if-else branches, because in CTP_ρ, each thread has a single control path. Throughout the transformation, trace statements in CTP_ρ are changed as follows:

1. Create unique names for local/shared variables in their definitions.

2. For each use of a local variable $v \in LV_i$, replace v with the most recent (unique) definition v'.

3. For each use of a shared variable $v \in SV$, create a unique name v' and add the definition $v' = \pi(v_1, \ldots, v_l)$ where each v_i, $1 \leq i \leq l$, is either the most recent definition of v in the same thread, or a definition of v in another concurrent thread. Then, replace v with the new definition v'.

From CSSA to Φ_{CTP_ρ}. Each event e is assigned a fresh integer variable $\mathcal{O}(e)$ denoting its execution time. Let $HB(e, e')$ denote that e happens before e' which is encoded as a logical

constraint: $\mathcal{O}(e) < \mathcal{O}(e')$. A path condition $g(e)$ is defined for each event e in CTP_ρ as follows such that e can be executed iff $g(e)$ is true:

- If e is the first event of a thread in the CTP_ρ then let $g(e) := \mathsf{true}$.
- Otherwise, let $e_1, \ldots, e_k$ be the sequence of thread-local events preceding e, and $g_{in} := \bigwedge_{i=1}^{k} g(e_i)$. Then,

$$g(e) = \begin{cases} c \wedge g_{in} & \text{if the statement of } e \text{ contains } \mathsf{assume}(c) \\ g_{in} & \text{otherwise} \end{cases}$$

Formula Φ_{CTP_ρ} consists of three main sub-formulas: $\Phi_{CTP_\rho} = \Phi^{PO}_{CTP_\rho} \wedge \Phi^{ST}_{CTP_\rho} \wedge \Phi^{\pi}_{CTP_\rho}$ where $\Phi^{PO}_{CTP_\rho}$, $\Phi^{ST}_{CTP_\rho}$, and $\Phi^{\pi}_{CTP_\rho}$ encode program order, statements, and π-Functions, respectively, and are constructed as following:

1. Let $\Phi^{PO}_{CTP_\rho} = \Phi^{ST}_{CTP_\rho} = \Phi^{\pi}_{CTP_\rho} = \mathsf{true}$, initially.
2. **Program Order:** For each event e in CTP_ρ with a thread-local preceding event e', let $\Phi^{PO}_{CTP_\rho} := \Phi^{PO}_{CTP_\rho} \wedge HB(e', e)$.
3. **Statements:** For each event e in CTP_ρ, if the corresponding statement of e has $lval := exp$, let $\Phi^{ST}_{CTP_\rho} := \Phi^{ST}_{CTP_\rho} \wedge (lval = exp)$. If e contains $\mathsf{assume}(c)$, let $\Phi^{ST}_{CTP_\rho} := \Phi^{ST}_{CTP_\rho} \wedge (g(e) \rightarrow c)$.
4. **π-Functions:** For each $w = \pi(v_1, \ldots, v_k)$, defined in e, suppose that e_i is the event that defines v_i. Let $\Phi^{\pi}_{CTP_\rho} := \Phi^{\pi}_{CTP_\rho} \wedge \bigvee_{i=1}^{k} [(w = v_i) \wedge g(e_i) \wedge HB(e_i, e) \wedge \bigwedge_{j=1, j\neq i}^{k} (HB(e_j, e_i) \vee HB(e, e_j))]$.

   Intuitively, the π-function evaluates to v_i iff it chooses the i^{th} definition in the π-set. Having chosen v_i, all other definitions occur before e_i or after the use of v_i.

Checking for Bugs. Formula Φ_{CTP_ρ} encodes all feasible linearizations of CTP_ρ. To check for a specific bug, e.g., an assertion or atomicity violation, another formula Φ_{bug} is built such that $\Phi_{CTP_\rho} \wedge \Phi_{bug}$ is satisfiable iff there is a linearization of CTP_ρ which leads to the bug.

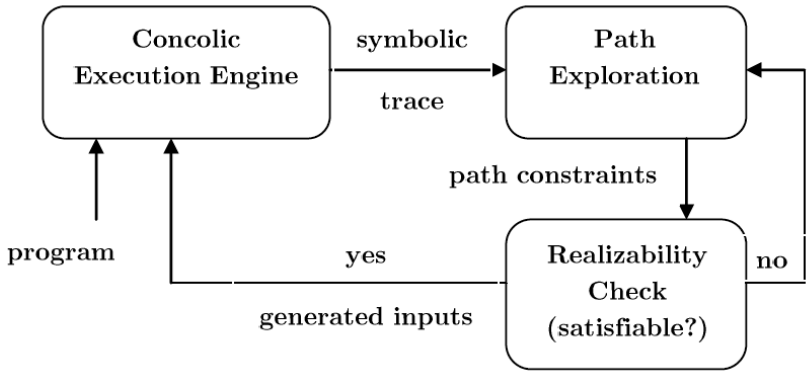


Figure 3.3: Concolic Testing.

3.2.4 Sequential Concolic Testing

Concolic testing is an effective test generation technique for sequential programs [24, 73, 5, 84, 4] for which different coverage criteria have been studied throughout the years. It assumes that the behavior of a sequential program solely depends on the values of inputs provided by the external environment, i.e., the program is deterministic. The main idea behind concolic testing is to use information available in previous executions of the program to generate input values that drive the execution towards covering uncovered parts of the program. It augments traditional symbolic execution with concrete execution by falling back upon concrete values observed during concrete execution to handle non-linear computations or calls to external library functions, for which no good symbolic representation is available.

As shown in Figure 3.3, concolic testing has three main components: concolic execution engine, path exploration, and realizability checker. Concolic execution engine executes the program with a given input vector concolically (i.e., executes the program with concrete and symbolic input values at the same time) and as a result, generates a symbolic trace that consists of a sequence of path constraints on symbolic inputs (i.e., branch conditions based on symbolic inputs encountered during execution). It generally fall back upon concrete values to handle non-linear computations or calls to external library functions. Different coverage criteria have been investigated and employed by concolic testing techniques. For example, path

coverage targets exploring all possible program execution paths, or control-flow coverage and its variations such as basic block coverage and explicit branch coverage target code coverage. These coverage criteria quantify a degree to which the program has been tested.

Given a symbolic trace, the path exploration component then selects one of the branch conditions and negates it (while keeping the previous branch conditions the same). The goal is to try to diverge from the already observed executions by taking a different side of an encountered branch. The path exploration component can follow a simple DFS or utilize some heuristics (e.g., branch statements, overall stack trace, the depth of the branches, and etc.) in selecting the target branch. Finally, the realizability checker component uses SMT solvers to generate an input vector (if possible) that would satisfy the new path constraints, with the understanding that such an input vector is likely to drive the execution of the program towards a different path.

3.3 Overview of Test Generation Using MTA

We target increasing branch coverage in concurrent programs using CTPs as under-approximation models for programs. To that end, each thread is subjected to sequential concolic testing, first, to increase branch coverage in individual threads. After each execution, essential information about the run (i.e., statements executed), current coverage (i.e., covered/skipped branches), and writes to shared variables in the execution is stored. Upon saturation, multi-trace analysis is used to generate new test inputs and thread schedules to cover previously uncovered branches.

The intuition behind this approach is that some of the bugs in concurrent programs might be sequential bugs that do not relate to any specific interleaving. The idea is to catch those bugs by sequential testing, which is cheaper than concurrent testing, without requiring to consider the interleaving space. Then, concurrent test generation aims to cover the remaining uncovered branches by exploring the input space and the interleaving space simultaneously to find a combination that would cause the branch to be taken.

Our MTA works as follows: First, we select a target branch of interest based on the current

coverage information. Then, we pick a stored run that has been previously observed to come *close* to the target branch but skipped it, i.e., the condition of the branch did not hold when the branch was hit. Suppose that the uncovered branch depends on a set of shared variables S. Generally, the branch condition may not be in terms of shared variables, but by intra-thread value tracking, we can obtain S.

Then, we choose candidate *interloper segments* from the set of so-far stored runs, such that the interloper segments contains a write to a shared variable in S. Note that these interloper segments may contain executions of multiple threads. The idea is to select an interloper segment and insert it (not necessarily atomically) into the runs that came close to the target branch, such that some of the shared variables on which the uncovered branch depends, are overwritten by the interloper segment before the branch condition gets evaluated.

We encode all possible interleavings where the interloper segment is inserted in the selected run and the target branch condition is satisfied as an SMT problem. Any solution to this problem implies input values and a schedule that covers the target branch.

3.4 Testing Algorithm

In this section, we first briefly discuss how we perform sequential testing of concurrent programs as the first step of our test generation technique. Then, we present our multi-trace analysis in detail.

3.4.1 Sequential Testing of Concurrent Programs

In order to perform sequential testing of a concurrent program, we first execute the program with a set of random inputs, I, to obtain a symbolic trace of the program (represented by ρ). Then, we focus on sequential testing of each thread T_i at a time. Based on the observed trace, we generate a trace ρ', which represents a sequential execution of T_i, by enforcing a set of ordering constraints between the events of different threads in ρ such that thread T_i is executed

sequentially and without any interference from other threads (if possible).

To do so, we generate happens-before relations on the events of ρ to enforce all of the events of other threads to happen after the last event of T_i in ρ. In cases where the complete sequential execution of T_i is not possible due to some synchronization, we stick to the corresponding orderings between the events of different threads in ρ to let T_i complete.

For sequential testing of thread T_i, we apply a traditional concolic testing technique [4] starting with input set, I, and following the schedule implied by ρ'. Then, we perform a depth-first search, making a path constraint (i.e., conjunction of the condition of the branches traversed) corresponding to the inner-most uncovered branch in T_i while requiring the condition of the uncovered branch to be true according to ρ'. A satisfiable solution for these constraints provides a set of inputs for the next round in concolic testing.

3.4.2 Multi-Trace Analysis (MTA) for Test Generation

Without loss of generality, we assume that there is an `if`-branch in thread T_i (whose condition depends on a shared variable x) which could not be covered by sequential concolic testing. Furthermore, we assume that there is a run rn which hits the corresponding `if`-statement while the condition of the `if`-statement is evaluated to `false`. The main goal is to generate a test (i.e., input values and a schedule) in which the last write to x before the `if`-branch in rn is overwritten by another write to x and the branch is covered. To that end, we find an interloper segment from a run (could be different from rn), with a write to x, that could be *soundly* (but not necessarily atomically) inserted after the last write to x in rn and search for possible input values and schedules by inserting the interloper segments into rn.

Algorithm 1 presents our test generation technique using MTA. The inputs of the algorithm include a concurrent program P, a set of branches Brs that are left uncovered during sequential testing, and a set of successful runs of the program Rns. Initially, Rns mostly contains sequential runs, but over time it accumulates multi-threaded executions as well. In fact, we extend the set of program runs (that form an under-approximation for the concurrent program)

Algorithm 1: CMTA(program P, branchSet Brs, runSet Rns)

```
1  while brs ≠ ∅ do
2      pick and remove br from Brs
3      Rns' ← set of runs from Rns hitting br
4      while Rns' ≠ ∅ do
5          pick and remove rn from Rns'
6          Vars ← set of variables affecting condition of br in rn
7          while Vars ≠ ∅ do
8              pick and remove var from Vars
9              (w, r) ← last write/read of var in rn before br
10             foreach event e such that w < e < r in rn do
11                 gLoc ← getGlobalLocation(rn, e)
12                 Segs ← findEnterloperSegments(gLoc, var, rns)
13                 while segs ≠ ∅ do
14                     pick and remove seq from Segs
15                     cs ← logicalEncoder(rn, br, var, r, seg, w, gLoc)
16                     if cs is satisfiable then
17                         extract input and schedule from the solution
18                         rn' ← P(input, schedule)
19                         Rns ← run ∪ {rn'}
20                         Brs ← brs ∪ getUncoveredBranches(rn')
```

For each uncovered branch, the algorithm goes over the runs hitting the branch and tries to find interloper segments from other runs (with a write of variable var affecting the branch condition) and insert it in the runs hitting the branch and search for input values and a schedule that cover the branch. The interloper segment should be inserted between the last write (w) and the last read (r) of var before the branch. getGlobalLocation returns global location at the insertion point, findEnterloperSegments returns interloper segments, logicalEncoder encodes the insertion problem as a logical constraint system. Any solution to the constraint system implies input values and a schedule that cover the branch. getUncoveredBranches returns yet uncovered branches according to the newly executed test run.

after each program execution correspondingly. The set of branches Brs is also updated with the branches skipped in the execution (if they have not been previously covered).

The main loop (lines 1-20) goes over the uncovered branches in Brs one by one and tries to generate a test that covers the target branch. For each uncovered branch br, it picks a set of runs Rns' from Rns that hit the branch condition. Obviously, the branch condition is `false` in all of these runs. Then, in lines 4-20, it iterates over these runs, searching for an appropriate interloper segment that could be inserted in the run. For each of these runs rn, it first finds the set of shared variables $Vars$ whose values affect the branch condition by performing a traditional def-use analysis on rn. Then, for each of these variables var, it tries to find a segment containing a write to var that can be inserted after the last write to var in rn (lines 7-20).

For a variable $var \in Vars$, let (w, r) be a pair of events where w denotes the last write to var before the target branch and r denotes the read of var just before the branch in rn. To break this write/read matching and overwrite the effect of w, the interloper segment should be inserted between w and r in rn. The interloper segments should be selected in such a way that they could be inserted soundly in rn. At a minimum, threads executing in the segment should be at the same locations as they are at the insertion point in rn, i.e., the global locations at the beginning of the segment and insertion point should be the same. The while loop at line 10, goes over the global locations at an event e in rn, such that $w < e < r$, where $<$ represents the order of the events in rn, and tries to find an appropriate set of interloper candidates.

Given a global location $gLoc$, a variable $var \in Vars$, and a set of runs Rns, algorithm findEnterloperSegments returns a set of segments from Rns such that the global location at the beginning of the segments is *consistent* with $gLoc$ and each segment contains a write to var. We discuss findEnterloperSegments (Algorithm 2) in detail, later. The while loop at line 13, goes over the interloper segments and calls logicalEncoder engine which generates a set of logical constraints encoding the set of all feasible runs of the program that result from inserting a specific segment seg at $gLoc$ in rn before the affecting read r such that the condition of br

is satisfied. We present logicalEncoder (Algorithm 2) in detail, later. Then, an SMT solver is used to check the satisfiability of the constraints. Any model satisfying the constraints implies a set of input values and a schedule that would cause the branch to be covered. In that case, the program is executed with the generated inputs according to the generated schedule to get a new run rn'. Then, rn' is added to Rns and the skipped branches in rn' are added to Brs if they are not covered previously (getUncoveredBranches returns such branches).

Finding Interloper Segments

Given a global location $gLoc$, a shared variable var, and a set of runs Rns, findEnterloperSegments (Algorithm 2) returns a set of segments from Rns such that the global location at the beginning of the segments is consistent with $gLoc$ and each segment contains a write to var. The while loop at line 3 goes over the runs in which there is at least one write to var. For each run, it iterates over the set of writes to var and finds candidate segments containing a write w to var and starting at a global location consistent with $gLoc$. Note that write w might be protected by some locks in the corresponding thread. In that case, the interloper segment should release all of those locks to let other threads be able to obtain the locks in the future without being blocked. Therefore, to build the interloper segment, the algorithm first moves forward to the first event ev after w in rn where the corresponding thread of w does not hold any locks. Let getFirstLockFreePoint(rn, w) return such event ev at line 7 (ev is equal to w when w is not a protected write).

Then, event ev is added to the segment and the algorithm moves backwards in rn, adding events to the segment, until it reaches w (lines 10-13). Then, it continues moving backwards in rn, considering the preceding events and adding them to the segment, until it reaches at a global location consistent with $gLoc$ (if possible). Note that there might be some threads not active in the segment. Requiring the location of such threads to match with $gLoc$ is too restrictive and could miss useful segments. Therefore, as we move backwards in rn, we keep track of the active threads in a set $Threads$. While moving backwards in rn, the algorithm

Algorithm 2: findEnterloperSegments(global location $gLoc$, variable var, runSet Rns)

```
1  segmentSet Segs ← ∅
2  Rns' ← set of runs in Rns that write to var
3  while Rns' ≠ ∅ do
4      pick and remove rn from Rns'
5      Wrts ← set of all writes to var in rn
6      foreach w ∈ Wrts do
7          ev ← getFirstLockFreePoint(rn, w)
8          seg ← ev
9          Threads ← {tid(ev)}
10         while ev ≠ w do
11             ev ← preceding event of ev in rn
12             seg ← ev.seg
13             Threads ← Threads ∪ {tid(ev)}
14         gLoc' ← getGlobalLocation(rn, ev)
15         while gLoc|Threads ≠ gLoc'|Threads and ev is not the first event in rn do
16             ev ← preceding event of ev in rn
17             gLoc' ← getGlobalLocation(rn, ev)
18             seg ← ev.seg
19             Threads ← Threads ∪ {tid(ev)}
20         if gLoc|Threads = gLoc'|Threads then
21             Segs ← Segs ∪ {seg}
22 return segs
```

Given a global location $gLoc$, variable var, and a set of runs Rns, the algorithm returns all interloper segments consisting of a write to var from Rns which start at a global location consistent with $gLoc$ and end at the first lock-free point after the write. getFirstLockFreePoint returns the first lock-free point after a given write in a given run. The algorithm then moves backwards in the run, until it reaches a global location consistent with $gLoc$ (if possible). $Threads$ keep track of the active threads in the interloper segment which are the only threads whose locations are required to match $gLoc$ at the beginning of the interloper segment.

searches for a global location $gLoc'$ where the projection of $gLoc'$ and $gLoc$ to the threads in $Threads$ are equal (lines 15-19); i.e., $gLoc|_{Threads} = gLoc'|_{Threads}$. If such global location is reached then the segment is added to the set of appropriate segments which is returned by the algorithm at the end.

Logical Constraint Encoding

Given a run rn, an uncovered branch br (which is skipped in rn), a shared variable var that affects the branch condition, an event r in run which reads from a shared variable var, an interloper segment seg, an event w in seg that writes to var, and a global location $gLoc$ in rn representing the insertion point, logicalEncoder generates a logical constraint system encoding the set of all feasible runs, in which the schedule is the same as in rn until reaching $gLoc$ and then events in the interloper segment are interleaved with the events in rn after global location $gLoc$ in a way that r is guaranteed to read the value written by w, and the condition of br is satisfied.

We call the event sequence in rn before the global location $gLoc$, the *prefix segment* and the event sequence after $gLoc$ and before br, the *main segment*. The inputs of the program are treated symbolically such that we could use SMT solvers [9, 10] to simultaneously search for input values and a schedule that would cause br to be covered. The SMT encoding is based on the concurrent trace programs (See Definition 3.2.4) of the main and interloper segments. However, unlike symbolic prediction, we consider symbolic values for inputs to be able to generate input values in addition to schedules.

Let CTP_{main} and CTP_{int} denote the CTPs of the main and interloper segments, respectively. Note that findEnterloperSegments ensures that the location of each thread, being active in the interloper segment, is the same at the beginning of both segments. Therefore, each active thread in the interloper segment should have a maximum common prefix of locations in both CTP_{main} and CTP_{int}. The thread may then diverge after this prefix in the segments.

Suppose that E^{main} and E^{int} represent the set of events in the main and interloper segments,

respectively. Note that not all of these events may be required for test generation. Indeed, certain events may be inconsistent with each other, e.g., if they originated from diverging runs. Therefore, for each event $e_i \in E^{main} \cup E^{int}$, an *indicator bit* b_{e_i} is considered whose value determines whether the event is required to happen before the target branch or not.

logicalEncoder generates a constraint formula Φ such that Φ is satisfiable iff there exist input values and a schedule (which follows the prefix segment and then interleaves the execution of threads in the main and interloper segments) that covers br. Φ consists of 7 different sub-formulas:

$\Phi = \Phi^{FP} \wedge \Phi^{PO} \wedge \Phi^{ST} \wedge \Phi^{\pi} \wedge \Phi^{BR} \wedge \Phi^{AWR} \wedge \Phi^{Ind}$.

Let $\Phi^{FP} = \Phi^{\pi} = \Phi^{Ind} = \mathsf{true}$, initially. Φ is constructed as follows:

1. Fixed Prefix (Φ^{FP}): For each event e_i in the prefix segment:

   - if e_i is the first event in rn, do nothing. Otherwise, let $\Phi^{FP} = \Phi^{FP} \wedge HB(e'_i, e_i)$ where e'_i is the predecessor of e_i in the prefix segment. This keeps the order of events the same as in the prefix segment.
   - if the corresponding statement of e_i has $lval := exp$, let $\Phi^{FP} = \Phi^{FP} \wedge g(e_i) \wedge (lval = exp)$. If e_i contains $assert(c)$, let $\Phi^{FP} = \Phi^{FP} \wedge g(e_i) \wedge (g(e_i) \rightarrow c)$. Note that $g(e_i)$ is required to be true in any case since all of the events in the prefixed segment are required to be enabled.

2. Program Order (Φ^{PO}): Let $\Phi^{PO} = \Phi^{PO}_{CTP_{main}} \wedge \Phi^{PO}_{CTP_{int}}$ where $\Phi^{PO}_{CTP_\rho}$ is as defined in Section 3.2.3. This reserves the order of events in both main and interloper segments.

3. Statements (Φ^{ST}): Let $\Phi^{ST} = \Phi^{ST}_{CTP_{main}} \wedge \Phi^{ST}_{CTP_{int}}$ where $\Phi^{ST}_{CTP_\rho}$ is as defined in Section 3.2.3. This encodes the statements in both main and interloper segments.

4. π-Functions (Φ^{π}): Define a new π-function for each shared variable use in $E^{main} \cup E^{int} - \{r\}$ to include definitions in the prefix, main, and interloper segments. As in standard

CTPs, for each $w = \pi(v_1, \ldots, v_k)$, let $\Phi^\pi = \Phi^\pi \wedge \bigvee_{i=1}^{k}[(w = v_i) \wedge g(e_i) \wedge HB(e_i, e) \wedge \bigwedge_{j=1, j\neq i}^{k}(HB(e_j, e_i) \vee HB(e, e_j))]$, where e_i denotes the event that defines v_i.

5. Target Branch (Φ^{BR}): Suppose that $(\mathsf{assume}(c), \emptyset)$ is the statement corresponding to the uncovered target branch. In rn, there is an event e_{br} that relates to the statement $(\mathsf{assume}(\neg c), \emptyset)$ corresponding to the other branch of the same conditional statement. Let $\Phi^{BR} = c \wedge g(e')$ where e' is the predecessor of e_{br} in the corresponding thread. This enures that the target branch is covered.

6. Affecting Write/Read Matching (Φ^{AWR}): Let W_{var} represent the set of all events that write to var in $E^{main} \cup E^{int}$. Then, $\Phi^{AWR} = HB(w, r) \wedge \bigwedge_{e_i \in W_{var}} (HB(e_i, r) \vee HB(w, e_i))$, guarantees that the read of var in r is matched the write to var in w.

7. Indicator Bits (Φ^{Ind}): For each event e_i in $E^{main} \cup E^{int}$:

   - $\Phi^{Ind} := \Phi^{Ind} \wedge (b_{e_i} \rightarrow g(e_i)) \wedge (b_{e_i} \rightarrow HB(e_i, e_{br})) \wedge (\neg(b_{e_i}) \rightarrow HB(e_{br}, e_i))$ saying that if b_{e_i} is true then its path condition should be true as well and it should happen before the target branch. Otherwise, it should happen after the branch.
   - If e_i belongs to thread T_i, let e_j be the predecessor of e_i in T_i. Then, $\Phi^{Ind} = \Phi^{Ind} \wedge (b_{e_i} \rightarrow b_{e_j})$ saying that each event happening before the branch requires that its preceding event (in the same thread) also happen before the branch.
   - Let $de_{T_i}^{main}$ and $de_{T_i}^{int}$ represent the first events of thread T_i after the common prefix in CTP_{main} and CTP_{int}, respectively. Since the segments diverge at de_{Ti}^{main} and de_{Ti}^{int}, for each thread after this point we should consider events either from the main segment or from the interloper segment; i.e., for each active thread T_i in the interloper segment: $\Phi^{Ind} = \Phi^{Ind} \wedge (b_{de_{T_i}^{main}} \rightarrow \neg(b_{de_{T_i}^{int}})) \wedge (b_{de_{T_i}^{int}} \rightarrow \neg(b_{de_{T_i}^{main}}))$.

Discussion on Scalability. As mentioned in Section 2.7, the main drawback of symbolic prediction (using CTPs) is that it would encounter scalability issues for large runs. The reason is that the SMT encoding is done at *statement* level which considers local computation as well as

global computation. As it symbolically encodes *all* possible interleavings of events of different threads, the size of the SMT problem grows rapidly as the size of CTP increases. Although our test generation technique here utilizes CTPs, the proposed SMT encoding is scalable to large CTPs. This is because normally, both the main and interloper segments are very small compared to the length of the runs. The SMT encoding keeps a large part of the generated schedule (i.e., the prefix segment) fixed and only encodes the interleavings of events in the main and interloper segments. This decreases the size of the generated SMT problem. We show this issue in our experiments.

3.5 Evaluation

We have implemented the test generation technique based using MTA on top of FUSION [90] - a symbolic prediction tool for multi-threaded C programs. To be able to generate inputs, FUSION is changed to consider symbolic values for input variables. We subjected our tool to a benchmark suite of multi-threaded C programs to evaluate the effectiveness of the testing technique. In the following, we briefly discuss the implementation and then present our experimental results.

3.5.1 Implementation

Figure 3.4 presents the architecture of the tool. It has four main components: a sequential concolic execution engine, a multi-trace analysis engine, an SMT solver, and a concurrent execution engine. Sequential concolic execution engine performs sequential testing of concurrent programs as discussed in Section 3.4.1. Multi-trace analysis engine consists of four sub-components: coverage-guided target selection, concurrent test run selection, interloper selection, and multi-trace SMT encoder. Coverage-guided target selection selects a target branch according to the number of attempts that have been made for covering that branch; branches with less number of attempts have priorities over the others. For a target branch, concurrent test

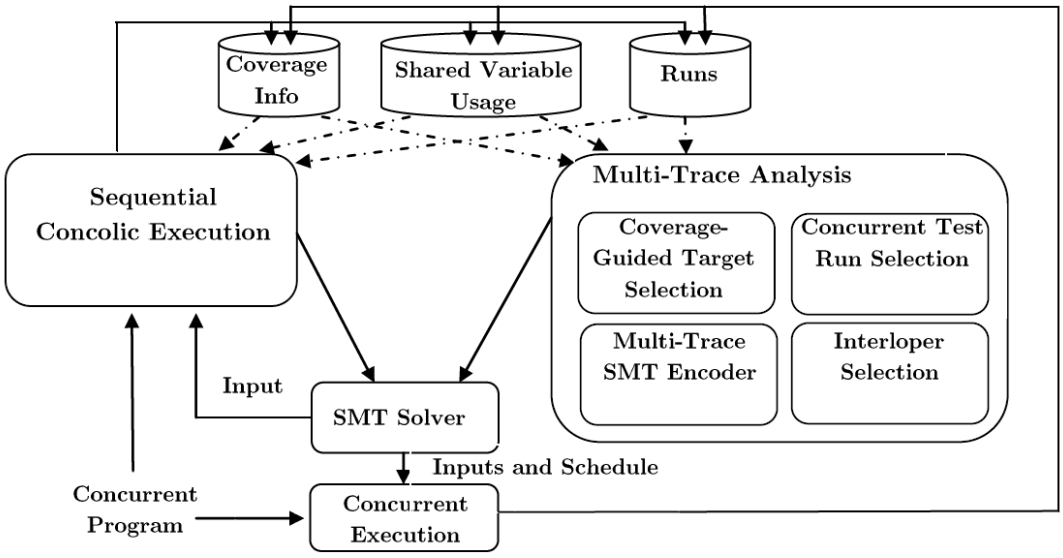


Figure 3.4: Test generation using MTA.

run selection returns a set of runs with different prefixes before skipping the branch. Interloper selection and multi-trace SMT encoder implement the algorithms presented in Section 3.4. Given a set of inputs and a schedule, concurrent execution engine executes the program with the inputs according to the schedule.

3.5.2 Experiments

We evaluated the effectiveness and efficiency of the test generation technique using MTA by subjecting the tool to a benchmark suite of multi-threaded C programs.

Benchmarks: `bluetooth` is a program based on a simplified model of bluetooth driver [62]. `apache1` and `apache2` are programs corresponding to two bugs in APACHE FTP server from *BugBench* [46]. `apache1s` and `apache2s` are simplified versions of `apache1` and `apache2`, respectively, where we removed parts of the code that were immaterial to branches with respect to shared variables. `ctrace` is a fast, lightweight trace/debug C library. `ctrace1` and `ctrace2` are two test drivers using this library, which contain some data races. `splay` is a program built using a C library implementing several tree structures. Finally, `aget` is a multi-threaded download accelerator. All of the benchmarks consist of two threads.

				Sequential Testing			Multi-Trace Analysis						
Program (LOC)	Num. of inputs	Thread	Num. of branches	Num. of tests	Num. of covered branches	Time (s)	Num. of tests	Num. of covered branches	Num. of branches with interlopers	Avg. num. of interlopers per uncovered branch	Time (s)	Bug found	% branch coverage seq.→conc.
bluetooth (88)	3	each	12	2	7	1	4	5	3	1	1	yes	58→100
apache1s (253)	1	each	8	4	7	3	1	1	1	2	8	yes	87→100
apache1 (640)	3	each	22	6	16	10	2	2	2	2	32	yes	72→81
apache2s (268)	2	each	10	3	7	2	2	2	3	1	5	yes	70→90
apache2 (864)	3	each	22	4	15	9	1	1	3	2	81	yes	68→73
ctrace1 (1466)	1-fixed	T1	64	1	35	17	1	3	8	2.3	341	yes	54→59
		T2	22	1	14	11	0	0	-	-	9	-	64→64
ctrace2 (1523)	1-fixed	T1	114	1	67	17	1	14	3	5	447	yes	59→71
		T2	22	1	14	11	0	0	-	-	9	-	64→64
splay (1013)	1-fixed	T1	16	1	5	5	3	7	3	2.3	90	no	33→75
		T2	16	1	11	10	1	1	2	2	75	bug	69→75
aget (680)	1-fixed	each	18	1	12	121	1	1	2	1	179	yes	66→72

Table 3.1: Experimental results for test generation using MTA

Table 3.1 presents the experimental results. We report the number of inputs of programs and for each thread in each program we show the total number of branches reported by FUSION. Note that according to the simplifications (e.g., constant propagation and etc.) applied on the observed traces by FUSION, the number of branches reported by FUSION is less than the actual number of branches in the program. For example, FUSION omits branches which depend only on local variables or relate to sanity checks on the system execution and does not include them in the total number of branches.

The table also contains information about the sequential testing of threads and the multi-

trace analysis. For sequential testing, we report number of generated tests, number of covered branches, and total time spent in testing each thread. For multi-trace analysis, we report number of generated tests, number of covered branches, total time, whether any bug is found, number of branches for which some interloper segments were found, and the average number of interloper segments found per branch. The table also presents the improvement in the percentage of branch coverage from sequential test generation to multi-trace analysis.

Observations: The experiments show that the sequential testing is able to cover a large number of branches quite fast. Note that some programs in our benchmark suite have fixed inputs. For example, `aget` expects a URL as input which we fixed for testing purposes. The sequential testing of these programs consists of a single execution of the program. As can be seen in the table, execution of these programs itself takes some considerable amount of time.

The experiments also show that MTA is successful in increasing branch coverage over sequential testing. For example, in case of `splay`, MTA increases branch coverage from 33% (in sequential testing) to 80%. According to the number of branches with interloper segments, we can see that for not many uncovered branches MTA could find interloper segments. However, often a test generated to cover an uncovered branch can lead to covering some other yet uncovered branches as well. Furthermore, trying only a few number of interloper segments were enough to find tests that cover such branches.

Another observation is that the total time spent on MTA is reasonable in practice. This is largely due (1) relying on the strength of sequential testing techniques to cover most branches sequentially, and (2) the effectiveness of MTA in finding interloper segments to cover target branches.

Furthermore, MTA is very effective in finding concurrency bugs. All of the bugs found in the benchmark suite were revealed by MTA by covering a branch that was not coverable in sequential testing. This suggests that branches that cannot be covered by purely sequential testing are good candidates for test generation.

Comparison with prediction techniques: Some programs in our benchmark suite have fixed

Bench-mark	FUSION on random inputs			FUSION on inputs generated by MTA		
	Runs	Time	bug found	Runs	Time	bug found
bluetooth	423	27s	no	33	4s	yes
apache1s	35	9s	no	2	1s	yes
apache1	399	3m4s	no	2	1s	yes
apache2s	126	2m2s	no	2	1s	yes
apache2	5974	23m50s	no	2	2s	yes
ctrace1	DNF			DNF		
ctrace2	DNF			DNF		
splay	30	27s	no bug	N/A		
aget	6	43s	yes	N/A		

Table 3.2: Comparing MTA with symbolic prediction using FUSION. DNF: did not finish, N/A: not applicable since inputs are fixed.

inputs. One can claim that predictive analysis may generate the same results in these benchmark. Note that MTA is different from predictive analysis (for which the inputs are fixed as well) in the sense that MTA aims to increase branch coverage (by exploring both input and interleaving spaces of the program) while predictive analysis does not perform input exploration and only explores the permutations of events of a single observed run.

To investigate the need for automated *input generation* for concurrent programs, we compared our tool with FUSION [90], based on our set of benchmarks. FUSION tries to find concurrency bugs such as data races. Since it does not report on coverage, here we only highlight whether the *known bugs* in the benchmarks are discovered by FUSION or not. In Table 3.2, we report the number of generated runs (i.e., schedules in this case) and the time spent by FUSION on prediction with both some *randomly generated* fixed inputs and some *bug-triggering* inputs generated by MTA.

We observe that for most of the benchmarks, FUSION cannot discover the bugs with random inputs. It can spend substantial analysis time in searching for alternative thread interleavings without finding the known bugs, as in the case of `apache2`. However, it performs pretty

well for some of the benchmarks with bug-triggering inputs generated by MTA. For `ctrace` benchmarks, FUSION did not finish due to the large overhead of symbolic prediction using CTPs which encodes all feasible permutations of events considering all computation (global as well as local) in the observed run. Although MTA also utilizes CTPs to generate tests, it was able to handle `ctrace` benchmarks, since as discussed before, often a large prefix of the generated schedule is fixed according to the runs in which interloper segments are inserted.

Conclusion: Our experiments showed that our MTA is very effective in increasing branch coverage in concurrent programs. Furthermore, using the MTA, we could find a large number of bugs in our benchmarks. This confirms that increasing branch coverage in concurrent programs is an appropriate approach for bug finding. We also compared our MTA with FUSION which is a symbolic prediction tool that explores the interleaving space with fixed inputs. Our experiments showed that applying FUSION with some randomly generated inputs on concurrent programs in most cases fails to find program bugs. However, our MTA analysis was able to find the bugs in the benchmarks as it performs input exploration as well as interleaving exploration. This shows the need for automated input generation in bug finding.

3.6 Related Work

Similar to MTA, some recent work [80, 79] use concurrent trace programs as approximation models of concurrent programs. These work target assertion violations, i.e, input/interleaving exploration is tailored towards finding assertion violations. The technique in [80] utilizes both over- and under-approximations of inter-thread communication on shared variables in concurrent trace programs to capture a suitable communication for finding assertion violations (in concurrent trace programs). In [79], a two-staged analysis is proposed which separates intra- and inter-thread reasoning. The first stage uses sequential program semantics to obtain a precise summary of each thread in terms of the accesses to shared variables made by the thread. The second stage performs inter-thread reasoning by composing these thread-modular sum-

maries using the notion of sequential consistency to find assertion violations. However, there are main differences between MTA and these work; MTA targets test generation, for exploring different program behaviours according to the uncovered parts of the program. Moreover, these techniques keep the under-approximation fixed and are restricted to concurrent trace program behaviours; e.g., they cannot check program assertions that do not show up in the concurrent trace program. MTA, on the other hand, extends the program approximation during the testing process. It uses program approximations to generate tests that target exploring parts of the program that do not appear in the approximation; i.e., it explores program behaviours that are beyond the approximation.

There are also some work exploiting analyses based on (*def-use*) pairs of shared variables in concurrent programs. For example, Shi et al. use invariants based on a def-use relation (obtained from a set of *bug-free* program runs) for bug detection and error diagnosis [76]. Wang et al. [91] follow a similar approach for coverage-guided testing. They utilize dynamic information collected from bug-free test runs to learn ordering constraints over the memory-accessing and synchronization statements. These ordering constraints are treated as likely invariants and are used to guide the selection of interleavings for future test execution. However, none of these techniques target structural coverage or perform input exploration.

Our notion of interloper segments is related to a work by Shacham et al. [75], where they consider testing linearizability [30] of concurrent collection operations by interleaving operations from a single adversarial environment. There, the search is guided towards interleaving non-commutative simple operations. However, interloper segments used by MTA may contain events from multiple threads, and are used to increase branch coverage in concurrent programs.

At a high level, our main insight to separate sequential coverage and leverage it for concurrent programs is similar to the insight by Joshi et al. [38], that many bugs in concurrent programs can be found by sequential analysis. Their goal, however, was to improve the usability of concurrent bug finding tools by filtering away bugs that can be reproduced purely sequential.

3.7 Summary

In this chapter, we proposed a test generation technique based on approximation models of concurrent programs. We used concurrent trace programs as approximation models and developed a multi-trace analysis to cover the uncovered part of the program based on the so far seen executions. More specifically, the analysis targets increasing branch coverage in concurrent programs. It combines information available in multiple runs of the program to (1) focus on an interloper segment of a run that provides values needed to take an uncovered branch, (2) insert the interloper segment in another run by searching for input values and an interleaving that would result in covering the uncovered branch. The multi-trace analysis encodes this problem as a set of logical constraints and uses SMT solvers to search for possible input values and interleaving. Our test generation technique utilizes the fact that the state-of-the-art sequential test generation techniques are generally able to quickly cover a large part of the program. Therefore, at the beginning, individual threads are exposed to sequential testing to increase branch coverage as much as possible. Upon saturation, our test generation technique falls back to the proposed multi-trace analysis to increase branch coverage. We implemented our technique in a tool that supports concurrent C programs. We performed some experiments that show the effectiveness of our multi-trace analysis in increasing branch coverage and finding concurrency bugs in concurrent programs.

Chapter 4

Bounded-Interference: A Heuristic for Providing Coverage Guarantees

Testing techniques for sequential programs are often coupled with a notion of coverage that the techniques provide. Different program coverage criteria (e.g., path coverage, branch coverage, etc.) have been introduced and targeted by sequential testing techniques. These coverage criteria quantify the effort put into the testing process in a meaningful way. For example, a sequential testing technique that provides branch coverage assures the tester that all of the branches in the program that could be covered under some input values are actually explored during the testing process.

Providing such coverage guarantees for concurrent programs is challenging since in addition to the input values, the exploration space is affected by the interleaving of execution of threads. Heuristics like context bounding [53, 54, 55] and delay bounding [12] were introduced and used by many techniques to reduce the exploration space into a manageable set that provides a meaningful coverage for concurrent programs. They characterize a subset of the search space by a bounding parameter p. As p is increased, more program behaviors are explored and in the limit it is guaranteed that all program behaviors are explored. For example, CHESS [53] is a tool from Microsoft that provides coverage guarantees on the interleaving space, i.e., all

interleavings up to a bounded number of context-switches are explored by keeping the inputs of the program fixed. Also, several sequentialization techniques [44, 42, 85, 62, 63] utilize the context bounding heuristic to search over both input and interleaving spaces (modulo context bound) for finding assertion violations in concurrent programs. The main problem with these heuristics is that they are defined based on the notion of control-flow among the threads (ignoring data completely) and therefore search strategies that utilize these heuristic are not guaranteed to be efficient. In fact, many thread interleavings might be equivalent to each other according to the way threads interfere with each other and therefore exploring all such interleavings imposes a huge overhead without exploring any new behaviour.

In this chapter, we introduce a new heuristic, called *bounded-interference*, for generating tests with coverage guarantees for concurrent programs. An *interference* happens whenever a thread reads a value from a shared variable which is provided by another thread. Based on the bounded-interference heuristic, the exploration space is limited to program behaviours within a bounded amount of interference among threads. A nice property of bounded-interference is that, since it is defined from the point of view of flow of data between threads (in contrast to the control-based notions such as context bounding), it can be very naturally incorporated into a setting in which the search for input values and interleaving can be performed in a unified manner. This heuristic might have applications beyond test generation; e.g., it can be used in model checking and program verification, bug localization, bug fixing, and etc. Here, we focus on the test generation aspect. Utilizing this heuristic, we have developed two different test generation techniques, based on concolic testing techniques, for concurrent programs, which are explained in Chapters 5 and 6 in detail.

4.1 Bounded-Interference Through An Example

An *interference* happens whenever a thread reads a value from a shared variable which is provided (i.e., written) by other threads. Bounded-interference heuristic is parameterized by the

```
1  public void Add(){
2    int status, pIO;
3    if (stoppingFlag)
4        status = -1;
5    else {
6        atomic{pendingIO++;}
7        status = 0;
8    }
9    if (status == 0){
10       assert(stopped==false);
11       //do work here
12   }
13   atomic{
14       pendingIO--;
15       pIO = pendingIO;
16   }
17   if (pIO == 0)
18       stoppingEvent = true;
19 }
```

```
20 public void Stop(){
21   int pIO;
22   stoppingFlag = true;
23   atomic{
24       pendingIO--;
25       pIO = pendingIO;
26   }
27   if (pIO == 0)
28       stoppingEvent = true;
29   assume(stoppingEvent==true);
30   stopped = true;
31 }
```

Figure 4.1: The simplified model of Bluetooth driver [62].

number of interferences among threads. The intuition behind it is to incrementally increase the degree of interference among threads while exploring program behaviours; i.e., first all program behaviours without any interference are explored. After that, all program behaviours with only one interference are explored. Then, all program behaviours with only two interferences are explored, and so on. In the following, we present the application of the bounded-interference heuristic for finding bugs in concurrent programs by an example.

Figure 4.1 shows a simplified model of the Bluetooth driver [62]. There are two dispatch functions, called `Add` and `Stop`. Function `Add` is called by the operating system to perform I/O in the driver and `Stop` is called to stop the device. There are four shared variables: `pendingIO`,

`stoppingFlag`, `stoppingEvent`, and `stopped`. The integer variable `pendingIO` is initialized to 1 and keeps track of the number of concurrently executing threads in the driver. It is incremented atomically whenever a thread enters the driver and is decremented atomically whenever it exits the driver. The boolean variable `stoppingFlag` is initialized to false and will be set to true to signal the closing of the device. New threads are not supposed to enter the driver once `stoppingFlag` is set to true. Variable `stoppingEvent` is initialized to false, and will be set to true after `pendingIO` becomes zero. Finally, `stopped` is initialized to false and will be set to true once the device is fully stopped; the thread stopping the driver sets it to true after it is established that there are no other threads running in the driver. Threads that call function `Add` expect `stopped` to be false (assertion at line 10) after they enter the driver.

Consider a concurrent program with two threads, T and T', calling `Add` and `Stop`, respectively. The assertion at line 10 in function `Add` ensures that the driver is not stopped before T starts working inside the driver. It is easy to see that this assertion always passes if T is executed sequentially, i.e., without any interference from T'. Therefore, if the assertion at line 10 is to be violated, it will have to be with some help from T', where a shared variable read in T reads a value written by a write in T'; we call these reads *non-local* reads.

We start by digressing *slightly* from the fully sequential execution of T, by letting *only one* read of a shared variable in T to be non-local. If the read from `stoppingFlag` at line 3 in T reads the value written by T' at line 22 then the `assert` statement at line 10 is not reachable since the `if`-branch at line 9 will not be covered. Selecting the read from `pendingIO` at line 6 in T as the non-local read, forces the read from `stop` in the assertion statement to read the initial value false (since only one read can be non-local), and hence the assertion check will be passed successfully. Finally, if we select the read from `stopped` in the assertion statement as the non-local read then it has to read the value written by T' at line 30. However, since both threads read and write to `pendingIO` at lines 6 and 24, there is no interleaving in which none of the reads from `pendingIO` is non-local. Therefore, the assertion cannot be violated by making only *one* read non-local. So, we digress more by allowing *two* reads of shared variables

to be non-local.

With two non-local reads, one can see that the assertion at line 10 can be falsified if the reads from `pendingIO` (at line 6) and `stopped` (at line 10) read the values written by T' at lines 24 and 30, respectively. A feasible interleaving that realizes this interference scenario would be the one in which T is executed first until it evaluates the branch condition at line 3, then thread T' is executed completely and then T continues execution.

4.2 Comparison with Context Bounding

The concept of context bounding was first introduced by Qadeer et al. [62] and later used by many techniques in test generation [53, 55], model checking [61], and sequentialization [44, 42, 85]. It characterizes a subset of program execution with a bounding parameter, i.e., the number of context-switches between threads. The idea is to incrementally increase this bound (starting from 0) and explore all program executions within the bounded number of context-switches. It is based on the conviction that most concurrency errors will be discovered within a small number of context-switches. In practice, the bound for the number of context-switches cannot go beyond 2 or 3 for real programs because of the large number of interleavings required to be explored even for a small bound. The main drawback of the context bounding heuristic is that it is defined based on the notion of control-flow among the threads and it completely ignores data-flow. In fact, many thread interleavings might be equivalent to each other according to the way threads interfere with each other and therefore exploring all such interleavings imposes a huge overhead without exploring any new behaviour.

Bounded-interference heuristic, on the other hand, is defined based on the notion of data flow among threads. All program executions with exactly the same set of interferences are behaviourally equivalent under the same input values, no matter how the execution of threads are interleaved. Therefore, for given input values it suffices to try *only one* of possible interleavings that realize each interference scenario. This property presents an advantage of

bounded-interference over context bounding; according to context bounding, all possible interleavings (no matter whether they are introducing new interference scenarios or not) with a bounded number of context-switches will be explored.

One interesting question is that whether there is any relation between the minimum bound required to catch a bug using bounded-interference and context bounding heuristics, respectively. Note that in both heuristics, all program behaviours will be explored in the limit, and therefore all program bugs will be caught by both heuristics in the limit. However, the problem is that the search space (even for a small bound) can be really large for real programs that makes it impossible to reach the limit. Theoretically, every bug that can be discovered using the bounded-interference heuristic with minimum bound k, appears on an execution with some k' number of context-switches, and therefore is discoverable using the context bounding heuristic with a bound at least as large as k'. However, there is no relation between k and k' in general; there are program bugs for which k is smaller than k' and program bugs in which k' is smaller than k. For example, the bug in the program presented in Section 4.1 requires at least 2 number of interferences and 2 number of context-switches according to bounded-interference and context bounding heuristics, respectively (i.e., $k = k'$). It is easy to imagine cases where k' is smaller than k; there a single context-switch can introduce several interferences. However, the cases where k is smaller than k' might not be so obvious in the first place. In the following, we present a buggy program which requires at least 3 context-switches but only one interference to reveal the bug.

Consider the buggy program in Figure 4.2. There are two threads, `Thread1` and `Thread2` where both have a batch update section in which they update a shared memory location `G` several times. Let us assume that according to the specification of the program, threads cannot be in their batch update section simultaneously. The input variable `turn` determines which thread can perform its updates first. Shared variables `start` and `done` identify the thread that just started and finished its updates, respectively. Each thread waits until `turn` indicates that it has permission to start its batch update (implemented by a while loop with a wrong condition).

```
1  public void Thread1(){
2    //entering batch update section
3    while(turn == 2);
4
5    //batch update
6    atomic{ started = 1; }
7    for(int i=0; i< 50; i++)
8      atomic{ G[i]=...; }
9
10   //finishing batch update
11   atomic{ turn = 2; }
12   atomic{ done = 1; }
13   assert(!(started==1 &&
14            turn==1 && done==1));
15 }
```

```
16 public void Thread2(){
17   //entering batch update section
18   while(turn == 1);
19
20   //batch update
21   atomic{ started = 2; }
22   for(int i=0; i< 50; i++)
23     atomic{ G[i+1]=...; }
24
25   //finishing batch update
26   atomic{ turn = 1; }
27   atomic{ done = 2; }
28   assert(!(started==2 &&
29            turn==2 && done==2));
30 }
```

Figure 4.2: A buggy implementation of accessing critical sections.

As soon as it gets the permission it updates `start` and enters the batch update section where it updates `G` for 50 times. After performing the updates, it gives the turn to the other thread to give it a chance to perform its updates (if it has not performed its updates yet) and updates `done` accordingly. An assertion then ensures that if the thread is the most recent one that started updating `G` and also the most recent one that finished its updates, then `turn` should be set for the other thread. Assume that the assertion statement is executed atomically.

However, the wrong condition of the while loops would cause input values other than 1 and 2 violate the specification requirement. This bug requires at least 3 context switches to be found, i.e., $k' = 3$. One execution that violates the assertion in `Thread1` would be the one in which `Thread2` is executed until it updates the `start` variable. Then, there is a context-switch and `Thread2` enters its batch update section (overwriting `start`), performs its batch updates, and updates the `turn` variable. Then, there is another context-switch and `Thread2` completes

its execution by performing its batch updates, overwriting `turn` and updating variable `done`. Then, another context-switch happens and `Thread1` continues execution by overwriting `done`. At this point, the assertion in `Thread1` is falsified and the bug is revealed. However, the assertion violation in `Thread1` can be discovered using the bounded-interference heuristic with the bound set to one (i.e., $k = 1$); it is enough to only consider the read from `turn` in the assertion as a non-local read that reads the value written by `Thread2` while finishing its batch updates. Note that any schedule that realizes this interference scenario still requires 3 context-switches, but this way, one can focus on finding a feasible schedule (e.g., encoding it as a SMT problem) realizing the interference scenario as opposed to exploring all interleavings with 3 or less number of context-switches.

However, none of the bounded-interference and context bounding heuristics is guaranteed to always perform better than the other in catching concurrency bugs. We believe that these heuristics can be viewed as two complementary heuristics and suggest that program analysis techniques use both of them side by side.

4.3 Bounded-Interference in Testing Concurrent Programs

The bounded-interference heuristic is defined based on the notion of data flow among threads. Therefore, it can be incorporated into the sequential concolic testing techniques to explore the interference scenario space (in addition to input exploration) to provide coverage guarantees for concurrent programs. We have developed two different techniques with coverage guarantees for concurrent programs, called *bounded-interference sequentialization* and *bounded-interference concolic testing*, by incorporating the bounded-interference heuristic into the sequential concolic testing. Here, we briefly discuss these two techniques. A detailed description of the techniques is provided in Chapters 5 and 6, respectively.

In bounded-interference sequentialization, we use the bounded-interference heuristic to transform concurrent programs into sequential programs, with the aim of being able to ap-

ply sequential concolic testing techniques to the generated sequential program *without any modification*. Bounded-interference sequentialization is parameterized with respect to a bound on the number of interferences; given a concurrent program and a bound k on the number of interferences, the sequentialization transforms the concurrent program into a sequential program such that every execution of the sequential program corresponds to an execution of the concurrent program with maximum k number of interferences. To be able to use the power of sequential concolic testing techniques (namely, input exploration) to explore both input and interference scenario spaces of concurrent programs, inputs of the generated sequential programs consist of the inputs of the corresponding concurrent program as well as some other inputs that specify interference scenarios (i.e., non-local reads and their matching writes). Therefore, by applying concolic testing on the generated sequential program, the input space and the interference space of the concurrent program will be explored systematically. The main advantage of this technique is that one can utilize of-the-shelf sequential concolic testing tools without any modification and benefit from their coverage-oriented search algorithms to generate tests for concurrent programs. Furthermore, it provides coverage guarantees (modulo interference bound) depending on the coverage guaranteed that the underlying sequential concolic testing tools provide. We present the bounded-interference sequentialization in Section 5 in detail.

Bounded-interference concolic testing, on the other hand, adapts sequential concolic testing techniques to support concurrent programs in the first place by incorporating the bounded-interference heuristic in their search algorithms. To that end, sequential concolic testing is equipped with one additional component, called *interference exploration* component, that enumerates all different interference scenarios in a systematic way. Bounded-interference concolic testing, exploits the fact that sequential concolic testing is able to quickly cover a large part of individual threads. Therefore, individual threads are exposed to sequential concolic testing first to increase coverage as much as possible in individual threads. Indeed, parts of the concurrent program that are not covered by sequential concolic testing may require some interferences among threads to be covered. After sequential concolic testing of threads,

bounded-interference concolic testing considers uncovered parts of the program and explores the interference space (modulo the interference bound) searching for input values and thread schedules that would result in covering an uncovered part of the program. The main advantage of this technique over bounded-interference sequentialization is that the exploration algorithm in bounded-interference concolic testing, provides coverage guarantees by its own and the completeness of the technique does not depend on the coverage guarantees of sequential concolic testing techniques. We present the bounded-interference concolic testing technique in Section 6 in detail.

Chapter 5

Testing Based on Bounded-Interference Sequentialization

To verify the effectiveness of the bounded-interference heuristic in finding concurrency bugs, we develop a sequentialization technique based on this heuristic. Several sequentialization techniques have been introduced in the literature [44, 42, 85, 62, 63] with the aim of reducing the problem of concurrent program analysis to sequential program analysis. Throughout the sequentialization, a concurrent program is transformed into a sequential program such that the sequential program embeds a subset of behaviours of the concurrent program. Then, available techniques for sequential program analysis can be utilized for analyzing the resulting sequential program (and hence the concurrent program). Most of the proposed sequentialization techniques utilize the context bounding heuristic to reduce the search space, i.e., the sequential program embeds a set of concurrent program behaviours within a bounded number of context-switches.

However, there are some problems with these sequentialization techniques that make it infeasible to apply traditional sequential testing techniques on the generated sequential programs: (1) The generated sequential program is highly non-deterministic. This is because it embeds a context-switch non-deterministically after each statement of the concurrent pro-

gram. Most sequential testing techniques (specifically those that provide coverage guarantees like concolic testing) assume that sequential programs are deterministic and hence they do not perform well on non-deterministic sequential programs. (2) Most of these sequentialization techniques [44, 42, 63] are aimed to be used in a *static* setting (where the programs are not executed) and for finding assertion violations. According to these sequentialization techniques if one wants to execute the generated sequential program, he has to guess the values of shared variables at the beginning of each context. As the result, the sequential program can get to unreachable states for wrong guesses. (3) The context bounding heuristic makes the exploration process inefficient, i.e., many thread interleavings might be equivalent to each other according to the way threads interfere with each other and therefore exploring all such interleaving imposes a huge overhead without exploring any new behaviour.

Because of the aforementioned problems, sequentialization techniques have been used *only* in *static* program analysis so far. Naturally, they suffer from static program analysis limitations; they do not perform well regarding memory tracking and calls to function libraries for which the source code is not available. They normally cannot handle complicated cases and also suffer from false positives (i.e., they might warn users in cases where the program is correct).

Our proposed sequentialization technique [64], called *bounded-interference sequentialization*, targets test generation and hence state-of-the-art sequential testing techniques (e.g., concolic testing [24, 73, 4]) can be applied on the resulting sequential program without any modification. This way, we can employ and benefit from the advanced exploration algorithms embedded into sequential testing techniques for an effective testing of concurrent programs.

Given a concurrent program P and an interference bound k, we propose a transformation that transforms P into a sequential program $\widehat{P}_k$ such that every execution of $\widehat{P}_k$ corresponds to at least one execution of P (might be a partial execution) in which there are at most k number of interferences. Our transformation effectively defers both the input generation and interference scenario selection tasks to the sequential testing technique, by encoding both as inputs to the newly generated sequential program. All program behaviours within a certain degree

of interference are encoded into the resulting sequential program, but the set of interferences is determined by the values of some inputs in the sequential program. Therefore, both input and interference spaces of the concurrent program P will be explored when the corresponding sequential program is subjected to sequential testing. We effectively encode all feasible interleavings for a set of interferences (defined by the inputs) as a set of constraints, and then use SMT solvers to ensure the soundness of our transformation.

Our concurrent program testing technique is then as follows: Each individual thread is exposed to sequential concolic testing (i.e., interference bound $k = 0$), first. Then, we incrementally increase k (starting with $k = 1$), sequentialize and perform sequential concolic testing on the resulting sequential program to find bugs. Applying a sequential testing technique with specific coverage guarantees would provide coverage guarantees (modulo the interference bound) on the concurrent program.

Our transformation has the following limitations: (i) It works for concurrent programs consisting of two threads. However, a study of concurrency bugs by Lu et al. [47] found that 96% of concurrency bugs involve only two threads. Therefore, the choice of limiting concurrent programs to contain only two threads should not be restrictive in finding concurrency bugs. (ii) It only allows one thread to be interfered by the other one. Our experiments show that this was not restrictive in finding concurrency bugs in our benchmarks, i.e., we could catch all of the known bugs in our benchmarks.

5.1 Preliminaries

In this section, we first fix the syntax of a simple sequential/concurrent programming language. We use it later, in Section 5.2, to present the transformation algorithm. We also define the notion of a *consistent global trace* which is used to prove the soundness of the sequentialization technique.

$< seq\_pgm >$ ::= $< input\_decl >< main\_method >$

$< input\_decl >$ ::= inputs: $< var\_decl >^*$

$< var\_decl >$ ::= int x; | bool x; | int[c] x; | bool[c] x;

$< main\_method >$ ::= main { $< var\_list >< stmt >$; }

$< var\_list >$ ::= vars: $< var\_decl >^*$

$< stmt >$ ::= $< stmt >; < stmt >$ | $< simple\_stmt >$ | $< complex\_stmt >$

$< simple\_stmt >$ ::= skip | $x =< expr >$ | assume($< b\_expr >$) | assert($< b\_expr >$)

$< complex\_stmt >$ ::= if ($< b\_expr >$) { $< stmt >$; } else{ $< stmt >$; }

$< expr >$ ::= x | c | $< b\_expr >$

$< b\_expr >$ ::= $true$ | $false$ | x | $\neg < b\_expr >$ | $< b\_expr > \vee < b\_expr >$

(a) Syntax of a simple sequential programming language.

$< conc\_pgm >$ ::= $< input\_decl >< var\_list >< init\_method >< seq\_pgm >^+$

$< init\_method >$ ::= init { $< stmt >$; }

$< complex\_stmt >$ ::= if ($< b\_expr >$) { $< stmt >$; } else { $< stmt >$; } |
lock (x) { $< stmt >$; }

(b) Syntax of a simple concurrent programming language.

Figure 5.1: Syntax of a simple sequential/concurrent programming language.

5.1.1 A Simple Sequential/Concurrent Programming Language

Here, we define the syntax of a simple sequential/concurrent programming language with variables, either scalars or arrays, ranging over integer and boolean domains (Figure 5.1). We assume that array sizes are specified statically during variable declaration. We also assume that programs are bounded, while loops are unrolled for a bounded number of times, and function calls are in-lined.

Figure 5.1a presents the syntax of a simple sequential programming. A sequential program

has a list of inputs and a method named $main$, from which it starts the execution. The $main$ method has a list of variables and a sequence of statements. Statements are either simple (e.g., skip, assignment, assume, and assert) or complex (e.g., conditional statement). Expressions can be integer constants, variables, or boolean expressions. Boolean expressions can be true, false, or boolean variables and can be combined using standard boolean operations. Furthermore, non-boolean expressions are implicitly transformed to boolean expressions in the natural way (i.e., false when the expression is evaluated to zero and true, otherwise) when assigned to boolean variables.

We define a concurrent program (Figure 5.1b) to be a finite collection of sequential programs (called threads) running in parallel. Threads share some variables, and their inputs are included in the inputs of the concurrent program. Here, definition of the complex statement is augmented by lock statements as a synchronization mechanism for accessing shared variables. A lock statement consists of a sequence of statements which are executed after acquiring a lock on a shared variable x. The semantics of locking mechanism is standard; whenever a thread obtains a lock on a variable, other threads cannot acquire a lock on the same variable unless the thread releases the lock. Each concurrent program has a method, named $init$, for initializing shared variables, and also for linking the inputs of the concurrent program to the inputs of the individual threads.

5.1.2 Global Traces (Revisited)

We defined the notion of a global trace in Section 2.2.1, which is a sequence of accesses to shared variables and synchronization operations. Note that a global trace does not contain any information about local computations (i.e., reads and writes to local variables). In this chapter, wherever we refer to program traces we mean global traces. However, we assume that program threads are created statically (as in Figure 5.1b) at the beginning of the program and there is no dynamic thread creation inside the threads.

Definition 5.1.1 (Consistent Global Traces)**.** *A global trace (Definition 2.2.1) is* consistent

if it is lock-valid (Definition 2.2.2), data-valid (Definition 2.2.4), and creation-valid (Definition 2.2.5).

Note that the programming language in Figure 5.1b does not allow dynamic thread creation (i.e., all of the threads are created statically at the beginning). Therefore, every global trace is creation-valid by default. As a result, the definition of consistent global traces are reduced to those that are lock-valid and data-valid.

Definition 5.1.2 (n-Interference Thread-Local Traces)**.** *A thread-local (i.e., all events correspond to the same thread) global trace α is a* n-interference thread-local trace *if there are n read events in α for which data-validity does not hold.*

According to the above definition, an n-interference thread-local trace contains n read events that read values different from what is written by the most recent write events to the corresponding variables.

Definition 5.1.3 (Feasible n-Interference Thread-Local Traces)**.** *Let P be a concurrent program with an input set I. A n-interference thread-local trace α for thread T_i is feasible if it corresponds to an execution of T_i under some input values $\overline{I}$ while allowing n reads from shared variables to read arbitrary values during the execution. We call such executions* interfered thread-local executions *under $\overline{I}$.*

Note that the above definition does not consider any restriction for the arbitrary values read by interfered read events. In fact, there might be no execution of P (under input values $\overline{I}$) with global trace ρ where $\rho|_{T_i} = \alpha$.

Lemma 5.1.4. *Let P be a concurrent program and ρ be a consistent global trace. If there exists some input values $\overline{I}$ such that for each thread T_i in P, $\rho|_{T_i}$ is a feasible n-interference thread-local trace (for some n) under $\overline{I}$ then ρ represents a feasible execution of P under $\overline{I}$.*

Proof. Since $\rho|_{T_i}$ corresponds to a thread-local execution of each thread T_i under $\overline{I}$ with n interference, therefore ρ respects program order (i.e., execution order in each thread). Moreover, ρ is data-consistent which shows that for each interfered read from a shared variable x

in a thread T_i there is another thread that can provide the value read by the interfered read for x. These facts together with lock-consistency of ρ guarantee that ρ corresponds to a feasible execution of P under $\overline{I}$. □

5.2 Sequentialization Algorithm

Let P be a bounded concurrent program consisting of two threads T and T', with a set of inputs I. Given an interference bound k in T (i.e., number of reads that are allowed to read values written by T'), we transform P into a sequential program $\widehat{P}_k$ such that each execution of $\widehat{P}_k$ corresponds to at least one (partial) execution of P with at most k interferences in T while T' is not interfered by T. The sequential program $\widehat{P}_k$ has an input set $\widehat{I}_k$ where $I \subset \widehat{I}_k$. The inputs in $\widehat{I}_k \setminus I$ specify the interference scenario (i.e., the set of non-local reads and their matching writes). Once k is fixed, there is a choice of which k reads to choose to get interfered in T and which k writes to choose as their corresponding writes in T'. Program $\widehat{P}_k$ takes all of these choices as inputs. This means that any sequential testing tool that explores the input space systematically will naturally try all possible interference scenarios (within the computation limit).

The sequential program $\widehat{P}_k$ has two copies of shared variables; each thread reads/writes on its own copy. $\widehat{P}_k$ first simulates the execution of T' to let it perform all writes that are supposed to provide values for non-local reads. Then, it simulates the execution of T where non-local reads read corresponding values produced by T'.

The sequential program $\widehat{P}_k$ simulates the execution of T' from the beginning until the first *lock-free point* (i.e., thread T' does not hold any lock), where all writes that are supposed to produce values for non-local reads, have occurred. The reason that the execution of T' is continued to a lock-free point (after all writes that provide values for non-local reads are performed) is that $\widehat{P}_k$ ultimately simulates the executions of P in which the events of T and T' are interleaved. Therefore, by moving to a lock-free point in T', we prune away unrealizable

executions in which T is blocked forever for acquiring locks that are never released by T'.

Since T' uses its own copy of shared variables during execution, the sequential program stores the values written by writes that are supposed to provide values for non-local reads, in some auxiliary variables and later loads these values while the corresponding non-local reads are being performed. When $\widehat{P}_k$ simulating the execution of T, it retrieves the value stored in the corresponding auxiliary variable as each non-local read is being performed.

Note that not all interference scenarios defined by inputs are realizable. Therefore, we have to ensure that there exists a feasible trace of P which (1) consists of the same events as in the execution of $\widehat{P}_k$, (2) observes for each interfered read in T the value written by the corresponding write in T', and (3) all other reads that are not involved in any interference read values written by their own thread (or the initial value of corresponding variable when there is no write to the variable before the read in that thread). To achieve this, all *global* events (i.e., accesses to shared variables and synchronization events) are logged during the execution of $\widehat{P}_k$, and a set of constraints is generated that corresponds to the existence of a feasible trace. Every time that T performs a read from a shared variable, we use a call to a SMT solver to check for the satisfiability of these constraints. If the feasibility check passes, it means that there exists a trace (representing an execution of P), with the same set of global events, in which the previous reads involved in interferences are reading from the writes defined by the interference scenario, and all other reads read from local writes. In this case, the execution of $\widehat{P}_k$ continues. Otherwise, the execution is abandoned to prevent exploring unreachable states. Note that since the interferences are limited to the ones specified by the inputs, the state of the program after passing each feasibility check is the same for any possible model of the constraint system. Therefore, it is just enough to ensure the existence of a feasible trace to be able to proceed the execution soundly. In the remainder of this section, we precisely define the transformation that was informally described here.

```
inputs: I;                                  main(){
        int[k] rds, wrts;                       //initialize G, G'
vars:   G, G';                                  ...
        int[k] vals;                            //read-write assumptions
        bool[k] rDone, wDone;                   ...
                                                τ[T'];
                                                assume(allWsDone());
                                                τ[T];
                                            }
```

Figure 5.2: Sequentialized program $\widehat{P}_k$.

5.2.1 Transformation Scheme

Figure 5.2 illustrates the sequential program $\widehat{P}_k$ generated based on concurrent program P consisting of threads T and T'. We assume that both T and T' are bounded sequential programs where loops are unrolled for a bounded number of times and function calls are in-lined. Therefore, all reads from shared variables in T and all writes to shared variables in T' can be identified and enumerated, according to their order in the corresponding thread. The input set of $\widehat{P}_k$ consists of `I` (i.e., inputs of the concurrent program P), and two arrays, `rds` and `wrts`, of size k specifying k interferences; `rds[i]` stores the identifier of the i^{th} non-local read in T and `wrts[i]` stores the identifier of the write in T' which is supposed to provide a value for `rds[i]`. In the case that the number of interferences is $k' < k$, we assume that `rds[i]` = `null` and `wrts[i]` = `null` for all $k' < i \leq k$.

The sequential program $\widehat{P}_k$ has two copies of shared variables, `G` and `G'`, on which T and T' operate, respectively. Variable `vals` is an array of size k, where `vals[i]` stores the value written by `wrts[i]`. There are also two arrays of size k, named `rDone` and `wDone`, such that `rDone[i]` and `wDone[i]` indicate whether the i^{th} non-local read and its matching write have occurred, respectively. All elements of these arrays are initialized to false. `wDone[i]`

and `rDone[i]` become true when `wrts[i]` and `rds[i]` are performed, respectively. These arrays are used to ensure that the corresponding reads and writes show up in the execution of $\widehat{P}_k$.

The `main` method in $\widehat{P}_k$ first initializes shared variables (according to the `init` method of concurrent program P). Then, it prunes away some obvious unrealizable interference scenarios (explained in the next paragraph). Then, it calls the transformed version of T' (represented by $\tau[T']$) and ensures that all writes specified by `wrts` have occurred during the execution of $\tau[T']$. This is done by `assume(allWsDone())` where function `allWsDone()` returns true if `wDone[i]` is true for all $1 \leq i \leq k$ (if `wrts[i]` is not `null`) and false, otherwise. Finally, it calls the transformed version of T (represented by $\tau[T]$).

As mentioned earlier, not all interference scenarios defined by `rds` and `wrts` are realizable. The minimum (but not sufficient) requirement is that for each non-local read `rds[i]` from a shared variable, its matching write `wrts[i]` should write to the same shared variable. This is ensured through a set of assumption statements in the `main` method. Note that in our transformation scheme, one always has the option of approximating the search space by allowing only a subset of reads in T to be non-local, and also by selecting only a subset of writes to the corresponding variable in T' as candidates for each non-local read. The `main` method also ensures that in the case that the number of interferences is less than k (say $k' < k$), then `rds[1..k']` and `wrts[1..k']` define the interference scenario and the rest of the elements in `rds` and `wrts` are set to `null`. This is done by a set of assumptions ((`rds[i]` = `null`) $\Rightarrow$ ($\bigwedge_{j=i+1}^{k}$`rds[j]` = `null` $\bigwedge_{j=i}^{k}$`wrts[j]` = `null`)) for $1 < i \leq k$. Note that we do not fix the number of interferences in advanced and it is defined by the `rds` and `wrts` inputs. Furthermore, to avoid exploration of redundant interference scenarios (where the same set of interferences are perturbed in `rds` and `wrts` arrays), the `main` method imposes an order on the the set of non-local reads such that `rds[i]` $<$ `rds[i+1]` for $1 \leq i < k$ (if `rds[i]` and `rds[i+1]` are not `null`) using assume statements.

In the following, we discuss the transformation of T and T'. The transformation uses

two auxiliary functions, `append` and `isFeasible`, to check for the existence of a feasible trace realizing the input interference scenario. Function `append` is used to add information about global events to a log file. Each global event is a tuple (T_i, a) where T_i is the identifier of the thread performing the event and a is a read/write action to a shared variable x or is a lock acquire/release action on a lock variable l. At any point during the execution of $\widehat{P}_k$, this log provides the exact sequence of global events that occurred up to that point. Function `isFeasible` checks whether the log can correspond to a *feasible* trace of program P (cf. Section 5.2.2).

Transformation Scheme for Interfering Thread T'

Figure 5.3 (on the right side) contains the transformation function τ for the statements of thread T'. The transformed program $\tau[T']$ is called by the `main` method and is executed until the first lock-free point (i.e., T' does not hold any locks) at which all writes specified in `wrts` have occurred. Note that the log contains all information necessary to determine which locks are held at any point in the execution. Function `returnCondition`, used in $\tau[T']$, returns true if T' is at a lock-free point and all writes in `wrts` are performed; otherwise, it returns false.

As mentioned before, T' operates on its own copy of shared variables, `G'`. However, to have a consistent log, for each shared variable access, we log an access to the corresponding variable in `G` instead of `G'`. For each shared variable x, let x' denote the corresponding copy for thread T' and let $(b\_)expr'$ be a (boolean) expression in which each shared variable x is replaced by x'.

For each expression, the transformation logs a read event from each shared variable read in the expression. For each assignment statement writing to a variable x, the right-hand side expression $(b\_)expr$ is transformed first and $(b\_)expr'$ is assigned to the corresponding variable; if x is a local variable then x is used as the left-hand side of the assignment, otherwise, x' is used as the left-hand side of the assignment to let T' work on its own copy of shared variables.

In case that the assignment statement writes to a shared variable, the transformation checks

$\mathcal{S}$ in T	Transformation $\tau[\mathcal{S}]$

$(b\_)expr$	//for each read r of x in $(b\_)expr$
	//where x is a shared var
	`if (`r `== rds[1]) {`
	`  x = vals[1];`
	`  rDone[1] = true;`
	`  append(log, (T, rd("x", x)), 1);`
	`  assume(isFeasible(log));`
	`} else if (`r `== rds[2]) {`
	`  x = vals[2];`
	`  assume(rDone[1]);`
	`  rDone[2] = true;`
	`  append(log, (T, rd("x", x)), 2);`
	`  assume(isFeasible(log));`
	`}`
	⋮
	`else if (`r `== rds[k]) {`
	`  x = vals[k];`
	`  assume(rDone[k-1]);`
	`  append(log, (T, rd("x", x)), k);`
	`  assume(isFeasible(log));`
	`} else {`
	`  append(log, (T, rd("x", x)));`
	`  assume(isFeasible(log));`
	`}`

`x =` $(b\_)expr$ (x is a shared var)	$\tau[(b\_)expr]$; `x =` $(b\_)expr$; `append(log, (T, wt("x", x)))`
$x = (b\_)expr$ (x is a local var)	$\tau[(b\_)expr]$; `x =` $(b\_)expr$

`lock(`x`){`$\mathcal{S}$`}`	`append(log, (T, ac(x)));`
	$\tau[\mathcal{S}]$;
	`append(log, (T, rel(x)))`

`assume(`b_expr`)`	$\tau[b\_expr]$; `assume(`b_expr`)`
`assert(`b_expr`)`	$\tau[b\_expr]$; `assert(`b_expr`)`

`if(`b_expr`){`$\mathcal{S}_1$`}`
`else{`$\mathcal{S}_2$`}`
	$\tau[b\_expr]$;
	`if(`b_expr`){`$\tau[\mathcal{S}_1]$`}`
	`else{`$\tau[\mathcal{S}_2]$`}`

$\mathcal{S}_1; \mathcal{S}_2$	$\tau[\mathcal{S}_1]$; $\tau[\mathcal{S}_2]$
`skip`	`skip`

$\mathcal{S}$ in T'	Transformation $\tau[\mathcal{S}]$

$(b\_)expr$	// for each read r from shared var
	// x in $(b\_)expr$
	`append(log, (T', rd("x", x)));`

`x =` $(b\_)expr$ (x is a local var)	$\tau[(b\_)expr]$; `x =` $(b\_)expr'$

`x =` $(b\_)expr$
(x is a shared var and w is the id of this write)
	$\tau[(b\_)expr]$;
	`x' =` $(b\_)expr'$;
	`if (`w `== wrts[1]) {`
	`  vals[1] = x';`
	`  wDone[1] = true;`
	`  append(log, (T', wt("x", x')), 1);`
	`  if (returnCondition()) return;`
	`} else if (`w `== wrts[2]) {`
	`  vals[2] = x';`
	`  wDone[2] = true;`
	`  append(log, (T', wt("x", x')), 2);`
	`  if (returnCondition()) return;`
	`}`
	⋮
	`else if (`w `== wrts[k]) {`
	`  vals[k] = x';`
	`  wDone[k] = true;`
	`  append(log, (T', wt("x", x')), k);`
	`  if (returnCondition()) return;`
	`}`

`lock(`x`){`$\mathcal{S}$`}`	`append(log, (T', ac(x)));`
	$\tau[\mathcal{S}]$;
	`append(log, (T', rel(x)));`
	`if (returnCondition()) return;`

`assume(`b_expr`)`	$\tau[b\_expr]$; `assume(`b_expr'`)`
`assert(`b_expr`)`	$\tau[b\_expr]$; `assert(`b_expr'`)`

`if(`b_expr`){`$\mathcal{S}_1$`}`	$\tau[b\_expr]$;
	`if(`b_expr'`){`$\tau[\mathcal{S}_1]$`}`
	`else{`$\tau[\mathcal{S}_2]$`}`

$\mathcal{S}_1; \mathcal{S}_2$	$\tau[\mathcal{S}_1]$; $\tau[\mathcal{S}_2]$
`skip`	`skip`

Figure 5.3: Transformation scheme for T and T'.

whether the write is in `wrts`. If the write is supposed to provide a value for the j^{th} non-local read (i.e., the write is equal to `wrts[j]`), the value of the shared variable is stored in `vals[j]` and `wDone[j]` is set to true. Then, a write event to x by T' is logged and function `returnCondition` is called to return when T' gets to an *appropriate* point.

For a lock statement on variable x, a lock acquire and a lock release event are logged right before and after the transformation of the lock body, respectively. Furthermore, after logging a lock release event function `returnCondition` is called to check whether the execution of T' should be stopped. Note that sequential programs do not have any lock statement. Therefore, here only the body of the lock statement is considered in the sequential program.

For assume and assert statements, the corresponding boolean expressions are transformed before these statements and the transformed statements refer to $(b\_)expr'$ instead of $(b\_)expr$. For each conditional statement, the transformation generates a conditional statement (having $(b\_)expr'$ instead of $(b\_)expr$ as the conditional expression) where the statements in both `if` and `else` branches are transformed, correspondingly. Here, the conditional boolean expression is transformed right before the conditional statement as well. The transformation of a sequence of statements includes the sequence of transformation of individual statements. Finally, skip statements stay unchanged.

Transformation Scheme for Interfered Thread T

Figure 5.3 (on the left side) presents the transformation function τ for the statements of thread T. The goal of transformation is to let each read of a shared variable in an expression in T be a candidate for a non-local read (observing a value provided by a write in T') while restricting the total number of non-local reads to k. When transforming a (boolean) expression, for each read r from a shared variable x we perform a case distinction:

(i) r is selected as one of the non-local reads by inputs; if r is the j^{th} non-local read, where $1 \leq j \leq k$, then x will read the value `vals[j]` written by `wrts[j]` in $\tau[T']$ and `rDone[j]` is set to true, indicating the j^{th} non-local read has been performed which is

required when the next non-local read (i.e., `rds[j+1]`) is performed. Then, a read event from x by T is logged and it records that it was the j^{th} non-local read. Finally, it ensures that a feasible trace exists that realizes the interference scenario so far. Therefore, it calls `isFeasible(log)` and stops the execution if no such feasible trace exists.

(ii) r is a local read, i.e., it does not belong to the input set `rds`. A read event from x by T is logged and then `isFeasible(log)` is called to ensure that a feasible trace in which this read and all previous local reads see values written locally (while all previous non-local reads are matched with the corresponding writes as specified by inputs) exists.

For each assignment statement, the right-hand side expression, is transformed (as discussed above), first. Then, the assignment comes unchanged. In case that the assignment writes to a shared variable, a write event to the corresponding variable is logged. For a lock statement on variable x, lock acquire and lock release events are logged right before and after the transformation of the lock body, respectively. Assume and assert statements remain the same unless the corresponding boolean expressions are transformed before these statements. For each conditional statement, the transformation generates a conditional statement (with the same conditional expression) where the statements in both `if` and `else` branches are transformed, correspondingly. Here, the conditional boolean expression is transformed right before the statement as well. The transformation of a sequence of statements includes the sequence of transformation of individual statements. Finally, skip statements stay unchanged.

5.2.2 Feasibility Check Constraints

The `isFeasible` function gets a log ρ as its input and checks for the existence of a feasible trace of the concurrent program (consisting of the events in ρ) in which `rds[i]` in ρ is reading from `wrts[i]` for $1 \leq i \leq k'$ (where $k' \leq k$ is the number of non-local reads appearing in ρ) and all other reads are reading values written by local writes. The `isFeasible` function generates a constraint system that encodes all such feasible traces and uses SMT solvers to

ψ: $\quad PO \wedge LC \wedge WRC_{interference} \wedge WRC_{local}$

PO: $\quad \bigwedge_{i=1}^{m-1} (t_{e_i} < t_{e_{i+1}}) \wedge \bigwedge_{i=1}^{n-1} (t_{e'_i} < t_{e'_{i+1}}) \wedge C_{init}$

C_{init}: $\quad (t_{init} < t_{e_1}) \wedge (t_{init} < t_{e'_1})$

LC: $\quad LC_1 \wedge LC_2$

LC_1: $\quad \bigwedge_{l \in L} \bigwedge_{\substack{[aq,rl] \in L_{T,l} \\ [aq',rl'] \in L_{T',l}}} (t_{rl} < t_{aq'} \vee t_{rl'} < t_{aq})$

LC_2: $\quad \bigwedge_{l \in L} \bigwedge_{\substack{aq \in NoRel_{T,l} \\ [aq',rl'] \in L_{T',l}}} (t_{rl'} < t_{aq})$

$WRC_{interference}$: $\quad \bigwedge_{1 \le i \le k} Coupled(rds[i], wrts[i])$

WRC_{local}: $\quad \bigwedge_{r \text{ not in } rds} Coupled(r, LocW(r))$

$Coupled(r, w)$: $\quad (t_w < t_r) \wedge \bigwedge_{w' \in W_x \setminus \{w\}} ((t_r < t_{w'}) \vee (t_{w'} < t_w))$

Figure 5.4: Constraints for checking the existence of a feasible schedule

find an answer. For each logged event e in ρ, an integer variable t_e is considered to encode the *timestamp* of the event. The constraints required for such feasible traces are captured using timestamps.

Figure 5.4 illustrates the constraint system. It is a conjunction of program order constraints (*PO*), lock-validity constraints (*LC*), write-read constraints for interferences ($WRC_{interference}$), and write-read constraints for local reads (WRC_{local}).

PO: Let $\rho|_T = e_1, e_2, ..., e_m$ and $\rho|_{T'} = e'_1, e'_2, ..., e'_n$ be the sequence of events in ρ projected to threads T and T', respectively. According to the program order, each event cannot happen unless its preceding event in that thread (according to ρ) has occurred. We also consider an initial event e_{init} which corresponds to the initialization of shared variables. This event should happen before any thread starts its execution in any feasible trace; it is encoded as the constraint C_{init} in Figure 5.4. The constraint *PO*, ensures that the order of events in T and T' is preserved.

LC: Each feasible trace should be lock-valid; i.e., threads cannot hold the same lock simultaneously. Each lock acquire event *aq* of lock l in the log is matched by precisely one lock

release event *rl* of lock l in the same thread, unless the lock is not released by thread T in the log. Each lock acquire event *aq* and its corresponding lock release event *rl* define a lock block, represented by $[aq, rl]$. Let $L_{T,l}$ and $L_{T',l}$ be the set of lock blocks of lock l in threads T and T', respectively. Then, LC_1 ensures T and T' cannot be inside lock blocks of the same lock l, simultaneously. Turning to locks that are not released by T in the log, the constraint LC_2 ensures that the acquire of lock l by thread T which is not released must always occur after all releases of lock l in thread T'. In this formula, $NoRel_{T,l}$ stands for lock acquire events in T with no corresponding lock release event.

***WRC*$_{interference}$ & *WRC*$_{local}$:** Let W_x represent the set of all write events to shared variable x in the log, and *LocW* be a function that for each read event r from a shared variable x returns the most recent write event to x in the log performed by the same thread; in case there is no such write event, e_{init} is returned. For each read event r from a shared variable x and write event w to the same variable, the formula $Coupled(r, w)$ ensures that r is coupled with w by forcing all events that write to x to happen either before w or after r. Therefore, *WRC*$_{interference}$ ensures that each read `rds[i]` is coupled with `wrts[i]` and *WRC*$_{local}$ ensures that all other reads are coupled with the most recent local writes to the corresponding variables.

5.3 Soundness and Completeness

Now, we discuss soundness and completeness of our testing technique based on bounded-interference sequentialization. Let P be a concurrent program with threads T and T', and $\widehat{P}_k$ be the corresponding sequential program which allows at most k interferences in T (while T' is not interfered by T).

Lemma 5.3.1. *Suppose that an error is reached in $\tau[T']$ when $\widehat{P}_k$ is executed with some input values $\overline{\texttt{I}}$, $\overline{\texttt{rds}}$, and $\overline{\texttt{wrts}}$. The error corresponds to an error in concurrent program P, i.e., there exists an execution of P which leads to the error.*

Proof. Let `log` be the log generated by $\widehat{P}_k$ during the execution. Since the error was revealed

in $\tau[T']$ and $\tau[T']$ simulates the execution of T' (on its own copy of shared variables) without any interferences from T, `log` should represent a consistent feasible n-interference thread-local trace of thread T' (for $n = 0$). Therefore, `log` corresponds the execution of P with input values $\overline{\texttt{I}}$ where T' is executed sequentially first, which leads to the error. □

Lemma 5.3.2. *Let `log` be a log (written by $\widehat{P}_k$ during the execution of $\widehat{P}_k$ with input values $\overline{\texttt{I}}$, $\overline{\texttt{rds}}$, and $\overline{\texttt{wrts}}$) at a feasibility check point. Suppose that the constraint system made by `isFeasible(log)` is satisfiable and ρ is a global trace obtained by ordering the events in `log` according to the timestamps satisfying the constraint system. Then, ρ is a feasible global trace of P (i.e., corresponds to an execution of P) under input values $\overline{\texttt{I}}$.*

Proof. $\texttt{log}|_T$ represents a feasible n-interference thread-local trace (for some $n \leq k$) and $\texttt{log}|_{T'}$ represents a sequential execution of thread T' (i.e., a feasible n-interference thread-local trace for $n = 0$) under input values $\overline{\texttt{I}}$. According to the lock-validity and write-read constraints in the feasibility check, ρ is a consistent global trace. According to the program order constraints $\rho|_T = \texttt{log}|_T$ and $\rho|_{T'} = \texttt{log}|_{T'}$. Therefore, both $\rho|_T$ and $\rho|_T$ are feasible n-interference thread-local traces. Based on Lemma 5.1.4, ρ should be a feasible global trace of P under input values $\overline{\texttt{I}}$. □

Lemma 5.3.3. *Suppose that an error is reached in $\tau[T]$ when $\widehat{P}_k$ is executed with some input values $\overline{\texttt{I}}$, $\overline{\texttt{rds}}$, and $\overline{\texttt{wrts}}$). The error corresponds to an error in concurrent program P, i.e., there exists an execution of P which leads to the error.*

Proof. Let `log` be the log generated by $\widehat{P}_k$ before revealing the error. There are two cases:

(i) The error is revealed before any non-local read. In this case, $\texttt{log}|_T$ is a consistent feasible n-interference thread-local trace of T (for $n = 0$). Therefore, $\texttt{log}|_T$ corresponds the execution of P with input values $\overline{\texttt{I}}$ where T is executed sequentially first which leads to the error.

(ii) The error is revealed after some non-local reads in $\tau[T]$. Let $\texttt{log}_{pre}$ be the prefix of `log` which is passed to the last call of function `isFeasible` in the execution of $\widehat{P}_k$

before the error. We know that $\texttt{isFeasible}(\texttt{log}_{pre})$ returns true since otherwise the execution would have been aborted. According to Lemma 5.3.2, there exists a (partial) execution R of program P under input values $\overline{\texttt{I}}$ with a global trace ρ such that $\rho|_T = \texttt{log}_{pre}|_T$ and $\rho|_{T'} = \texttt{log}_{pre}|_{T'}$. Let R' be an execution of P under $\overline{\texttt{I}}$ where partial run R is continued by executing T and let ρ' be the global trace according R'. $\texttt{log}_{pre}|_T$ is a common prefix of $\texttt{log}|_T$ and $\rho'|_T$. On the other hand, $\texttt{log}|_T$ and $\texttt{log}_{pre}|_T$ are feasible n-interference thread-local traces of T (for some n) under $\overline{\texttt{I}}$ with exactly the same interfered reads. Therefore, $\rho'|_T = \texttt{log}|_T$ and the error is revealed in T in R'.

□

Theorem 5.3.4. *Every error revealed in the execution of $\widehat{P}_k$ corresponds to an error in the concurrent program P, i.e., there exists an execution of P that leads to the error.*

Proof. The error is either revealed in $\tau[T]$ or in $\tau[T']$. According to Lemmas 5.3.1 and 5.3.3, in either case there exists an execution of P which leads to the error. □

Now, we discuss the coverage guarantees of bounded-interference sequentialization, based on the coverage guarantees of the backend sequential testing tools.

Lemma 5.3.5. *Let $\widehat{P}_k$ be the sequential program obtained from a concurrent program P according to the transformation presented in Section 5.2. Suppose that a bug requires k' interferences (where $0 \leq k' \leq k$) in thread T and no interference in thread T' to be revealed under some input values in P. Then, there exists some input values $\overline{I}$, $\overline{\texttt{wrts}}$ and $\overline{\texttt{rds}}$ for $\widehat{P}_k$ such that the execution of $\widehat{P}_k$ with these inputs reveals the bug.*

Proof. Let ρ be the global trace corresponding to an execution R of program P with k' interferences in T under input values $\overline{I}$ that reveals the bug. There are two cases:

- $k' = 0$, i.e., the bug is a *sequential* bug either in T or in T'. Let `rds[i]` = `null` and `wrts[i]` = `null` for all $i \leq k$. We claim that the execution of $\widehat{P}_k$ under input values $\overline{I}$, $\overline{\texttt{wrts}}$ and $\overline{\texttt{rds}}$ reveals the bug. Let `log` be the log generated during the execution of $\widehat{P}_k$.

First, assume that the bug is in T'. $\rho|_{T'}$ represents a feasible 0-interference global trace for sequential execution of T' under input values $\overline{I}$. On the other hand, $\widehat{P}_k$, calls $\tau[T']$ which simulates the sequential execution of T' (on its own copy of shared variables) under input values $\overline{I}$. Furthermore, since `wrts[i]` = `null` for all $i \leq k$, the execution of $\tau[T']$ will not be stopped by any `return` statement added in $\tau[T']$ (to return after the first lock-free point where all writes in `wrts` are performed). Therefore, $\texttt{log}|_{T'} = \rho|_{T'}$ which reveals the bug.

Now, assume that the bug is in T. $\rho|_T$ represents a feasible 0-interference global trace for sequential execution of T under input values $\overline{I}$. $\widehat{P}_k$ first calls $\tau[T']$ which works on the second copy of shared variables. Therefore, when $\tau[T']$ returns, the main copy of shared variables have their initial values. Since `rds[i]` = `null` for all $i \leq k$, each from a shared variable reads a value generated locally during the execution of $\tau[T]$ and hence $\widehat{P}_k$ simulates the sequential execution of T under input values $\overline{I}$. Therefore, $\texttt{log}|_T = \rho|_T$ which reveals the bug.

- $k' > 0$, i.e., the bug is a concurrency bug that is revealed in T. From ρ, we obtain a set of input values for $\widehat{P}_k$ as follows:
  Let $\gamma = \{(id(r), id(w)) \mid r$ is a read event by T and w is its matching write performed by T' in $\rho\}$ where function id returns the identifier of the corresponding read or write; i.e., γ specifies the set of interferences in ρ. We sort γ such that $(id(r), id(w)) < (id(r'), id(w'))$ iff $id(r) < id(r')$. Let `rds[1..k']` = $Reads(\gamma)$ and `wrts[1..k']` = $Writes(\gamma)$ where $Reads$ and $Writes$ return arrays consisting of the reads and writes involved in γ in the sorted order. Let `rds[k'+1..k]` = `null` and `wrts[k'+1..k]` = `null`. We claim that $\widehat{P}_k$ reveals the bug when it is executed with input values $\overline{I}$, $\overline{\texttt{wrts}}$ and $\overline{\texttt{rds}}$:

  Let `vw[1..k']` and `vr[1..k']` represent the values written by `wrts[1..k']` or read by `rds[1..k']` in ρ, respectively. Assume that the execution of $\widehat{P}_k$ with input values $\overline{I}$,

$\overline{\texttt{wrts}}$ and $\overline{\texttt{rds}}$ generates a log file `log`. We prove that $\texttt{log}|_T = \rho|_T$ which reveals the bug.

$\widehat{P}_k$ first calls $\tau[T']$. The execution of $\tau[T']$ with input values $\overline{I}$, $\overline{\texttt{wrts}}$ and $\overline{\texttt{rds}}$, simulates sequential execution of T' with $\overline{I}$. We know that $\rho|_{T'}$ represents a feasible 0-interference global trace for sequential execution of T' with input values $\overline{I}$ that contains `wrts[1..k']`. Therefore all writes in `wrts[1..k']` occur in the execution of $\tau[T']$ as well and `vals[1..k'] = vw[1..k']` by the time that $\tau[T']$ is returned.

$\widehat{P}_k$ calls $\tau[T]$ after $\tau[T']$. Let $\texttt{log}|_{T,i}$ and $\rho|_{T,i}$ represent prefixes of $\texttt{log}|_T$ and $\rho|_T$ right before the i^{th} non-local read, respectively. $\rho|_{T,1}$ corresponds to sequential execution of T under input values $\overline{I}$ until the first non-local read. Also, before reaching the first non-local read, $\widehat{P}_k$ simulates the sequential execution of T under input values $\overline{I}$. Therefore, `rds[1]` will occur in the execution of $\widehat{P}_k$ and $\texttt{log}|_{T,1} = \rho|_{T,1}$. When reaching `rds[1]`, $\widehat{P}_k$ loads value `vals[1] = vw[1]` into `rds[1]`. Therefore, $\widehat{P}_k$ continues simulating the same thread-local execution path of T as the thread-local execution path of T in R, until reaching `rds[2]`. As a result $\texttt{log}|_{T,2} = \rho|_{T,2}$. The same kind of argument is valid for later non-local reads and therefore $\texttt{log}|_{T,k'} = \rho|_{T,k'}$. When reaching `rds[k']`, $\widehat{P}_k$ loads value `vals[k'] = vw[k']` into `rds[k']`. After `rds[k']` is performed, $\widehat{P}_k$ continues simulating the same thread-local execution path of T as the thread-local execution path of T in R (i.e., $\texttt{log}|_T = \rho|_T$), which reaches the error. Note that according to ρ, $\texttt{isFeasible}(\texttt{log})$ is guaranteed to return true wherever it is called during the execution of $\widehat{P}_k$. Therefore, the execution of $\widehat{P}_k$ will not be aborted before getting to the error.

□

Theorem 5.3.6. *Suppose that a testing tool provides path coverage guarantees. Let $\widehat{P}_k$ be the sequential program obtained from a concurrent program P according to the transformation presented in Section 5.2. Subjecting the testing tool to $\widehat{P}_k$ will catch all bugs that require k'*

interferences (where $0 \leq k' \leq k$) in thread T and no interference in thread T' to be revealed under some input values in P.

Proof. Assume that a bug requires k' interferences (where $0 \leq k' \leq k$) in thread T and no interference in thread T' to be revealed under some input values in P. According to Lemma 5.3.5 there exists some input values $\overline{I}$, $\overline{\texttt{wrts}}$ and $\overline{\texttt{rds}}$ such that the execution of $\widehat{P}_k$ under these inputs reveals the bug. Therefore, there exists at least one execution path in $\widehat{P}_k$ that leads to the bug. Hence, a sequential testing tool that provides path coverage guarantees should be able to catch the bug. □

5.4 Evaluation

We have implemented a prototype testing tool for multi-threaded C# programs according to the bounded-interference sequentialization and applied it on a benchmark suite. In the following, we first briefly discuss the implementation and then present the experimental results.

5.4.1 Implementation

Figure 5.5 presents a high-level view of testing based on bounded-interference sequentialization. We used a C# parser, CSPARSER[1], for source-to-source transformation. We changed the parser such that in addition to a concurrent program, it gets an unrolling bound (u) for loops and an interference bound (k) as inputs and transforms the concurrent program to a sequential program, according to the transformation rules presented in Section 5.2. Throughout the transformation, each loop in the concurrent program is unrolled for u times, and all reads and writes to shared variables are identified and enumerated. In this prototype, the assumptions in the `main` method of the sequential program (defining the set of possible interference scenarios), are generated manually.

[1] http://csparser.codeplex.com/

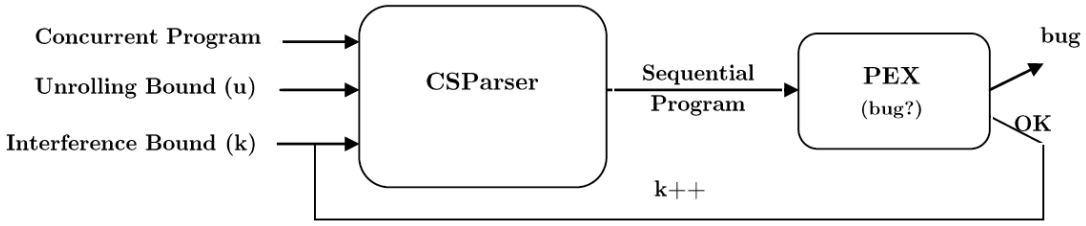


Figure 5.5: Test generation by bounded-interference sequentialization.

According to Theorem 5.3.6, the testing technique is complete if the resulting sequential program is subjected to a sequential testing tool that provides path coverage guarantees. However, providing path coverage guarantees can be very expensive in practice. We used Microsoft PEX [84] as our backend sequential testing tool which itself uses Z3[2] [9] as the underlying SMT solver. PEX targets different variations of control-flow coverage such as basic block coverage, explicit and implicit branch coverage. Control-flow coverage, in general, is weaker than path coverage in the sense that it might miss some program bugs. However, PEX managed to find all known bugs and some new bugs in our benchmarks.

For a given concurrent program, we sequentialize the program, starting with interference bound $k = 1$, and use PEX to test the resulting sequential program. In the case that no error is found by PEX, the interference bound is increased incrementally and the same process is repeated until the time/computation limit is hit, an error is found, or all possible interference scenarios are explored.

5.4.2 Experiments

Benchmarks: To evaluate the effectiveness of our tool, we performed some experiments on a benchmark suite of C# programs. `Bluetooth` is simplified model of the bluetooth driver presented in Figure 4.1. `Account` is a program that creates and manages bank accounts. `Meeting` [44] is a sequential program for scheduling meetings. Like in [44], we assumed that there are two copies of the program running concurrently. `Vector`, `Stack`, `StringBuffer`,

[2]http://research.microsoft.com/en-us/um/redmond/projects/z3/

program (LOC)	Num. of bugs (1 interference)	Num. of bugs (2 interferences)	Num. of bugs (3 interferences)	Num. of tests generated by PEX	total time (sec.)
`Bluetooth` (55)	0	1	0	18	26
`Account` (103)	1	0	0	38	28
`Meeting` (101)	1	0	0	12	16
`Vector1` (345)	0	1	0	55	104
`Vector2` (336)	0	1	0	51	80
`Vector3` (365)	0	1	0	82	102
`Stack1` (340)	0	1	0	52	100
`Stack2` (331)	0	1	0	49	74
`Stack3` (361)	0	1	0	79	98
`HashSet` (334)	1	0	0	32	22
`StringBuffer` (198)	1	0	0	12	12
`Series` (230)	1	0	0	9	10
`SOR` (214)	0	1	0	372	490
`Ray` (1002)	1	0	0	7	18
`FTPNET` (2158)	2*	0	0	10	56
`Mutual` (104)	1	0	0	28	10

Table 5.1: Experimental results for testing based on bounded-interference sequentialization. Symbol * means new bugs found.

and `Hashset` are all classes in Java libraries. To test these library classes, we wrote programs with two threads, where each thread executes exactly one method of the corresponding class. `Series`, `SOR`, and `Ray` are Java Grande multi-threaded benchmarks[3]. For the above Java programs, we used a Java to C# converter to transform the corresponding Java classes to C#. `FTPNET`[4] is an open source FTP server in C# and `Mutual` is a buggy program (presented in Figure 4.2) in which threads can be in a critical section simultaneously due to improper synchronization.

We set the loop unrolling bound u to 2 and sequentialized the programs for interference bound $1 \leq k \leq 3$ (there was no interference scenario with $k \geq 4$ interferences in one thread T

[3]http://www.javagrande.org/
[4]http://sourceforge.net/projects/ftpnet/

in any of the benchmarks). In the `main` method of the sequential program, we added assumptions allowing *all* possible interference scenarios; i.e., we let each shared variable read in T to be a non-local read and consider all writes to the corresponding shared variable in thread T' as possible matches for the read.

Table 5.1 contains information about the number of bugs found by allowing k interferences for $1 \leq k \leq 3$, the total number of tests generated by PEX, and the total time spent by PEX for testing the generated sequential programs for each benchmark.

Observations: The experiments show that the bounded-interference heuristic is very effective in finding concurrency bugs. All of the bugs were found by allowing only one or two interferences. This implies that concurrency bugs are not very complex, normally. In all of the benchmarks, no new error was found when k was increased to 3. Since these benchmarks have been used by other tools before, we know of no (previously found) bugs that were missed by our tool. Moreover, we found some new bugs that were not previously reported in `FTPNET`.

Another observation is that the total number of generated tests by PEX is reasonable. Since this number is directly affected by the total number of possible interference scenarios (in which one thread is interfered by the other one), we can conclude that the bounded-interference heuristics performs pretty well in reducing the search space while being effective in finding bugs.

Furthermore, the testing technique is very time efficient; the testing time for all of the benchmarks (except `SOR`) was less than two minutes. For `SOR`, the majority of time (about 7 minutes) was spent for testing the program with 3 interferences. This is because there are many shared variables reads in `SOR` and many options for the coupling writes for each read.

Although the main goal of the sequentialization was to quickly test whether the bounded-interference heuristic performs well in practice, we were curious to see how it performs compared to the other sequentialization techniques. We selected POIROT[5], which is a tool that exploits context-bounding sequentialization of concurrent programs and performs a static anal-

[5]http://research.microsoft.com/en-us/projects/poirot/

ysis to find *assertion violations*, and compared our tool with it. However, a side by side comparison with POIROT, when it does not aim for test generation to increase code coverage, is not meaningful.

POIROT has its own input language. We picked two of the benchmarks (i.e., `SOR` and `Mutual`), which did not use the object oriented paradigm and could be transformed to the the input language of POIROT more naturally, and translated them manually. Our experiments showed that POIROT did not scale well for these benchmarks. POIROT failed to catch the bug (which is an assertion violation) in `SOR` for context bound of 2, 3, and 4 within 30 minutes (for each bound). For `Mutual`, which requires 3 context switches to expose the bug, we set the loop unrolling bound $u = 50$. Our tool found the bug in a few seconds while Poirot failed to catch the bug for context bound of 3 within 30 minutes.

Conclusion: Our experiments showed that the bounded-interference heuristic is very effective in finding concurrency bugs. In fact, all of the bugs in our benchmarks were found by allowing only a few number of interferences among threads. This suggests that the bounded-interference heuristic can be used by test generation techniques to efficiently search through the exploration space. We also performed some experiments to compare our testing technique based on the bounded-interference sequentialization with a POIROT that uses the context bounding sequentialization to find bugs. Our experiments showed that the bounded-interference heuristic is more efficient than the context bounding.

5.5 Related Work

The idea of sequentializing concurrent programs and analyzing the resulting sequential programs was first proposed by Qadeer et al. in [62]. According to their sequentialization technique, the generated sequential program simulates the behaviours of the concurrent program up to only *two* context-switches. Their transformation algorithm is cheap in the sense that it does not introduce any additional copies of the shared variables. However, allowing maximum

two context-switches is too restrictive for catching many concurrency bugs.

Lal and Reps [44] proposed another sequentialization technique in which a boolean concurrent program with finitely many threads is transformed to a boolean sequential program which encodes the behaviours of the concurrent program corresponding to k rounds of executions of threads (in a round-robin manner). Their transformation introduces k extra copies of shared variables, one to keep the values of shared variables at each round. The sequential program calls the threads sequentially, and each thread uses the corresponding set of shared variables in each round. It then ensures that the values of shared variables at the end of each round are equal to the values of them at the beginning of the next round. Lahiri et al [42] adapted the sequentialization technique of Lal and Reps for C programs. Appealing to this transformation, Rakamaric implemented a tool, called STORM [63], for static unit checking. Later, La Torre et al. [85] adjusted the sequentialization technique of Lal and Reps for k context-switches instead of k context-rounds. However, all of these sequentialization techniques are meant to be used in static program analysis; execution of the sequential programs requires guessing the values of shared variables at the beginning of each context/round which might lead to unreachable states for wrong guesses.

La Torre et al [85] also proposed a lazy sequentialization technique using the context bounding heuristic that does not introduce any additional copies of shared variables. The idea is to execute the active thread in each context from the beginning to re-compute the values of local variables at the beginning of that context while using the corresponding pre-computed values of shared variables at the beginning of each previous context in which the thread was active. The main problem with the lazy transformation is that it has a huge overhead calling threads multiple times.

Another common problem with all of the aforementioned sequentialization techniques is that the generated sequential programs are highly non-deterministic; a context-switch is added after each statement of the concurrent program non-deterministically in the sequential program. Therefore, it is not feasible to apply sequential testing techniques on the generated sequential

programs. In contrast, the bounded-interference sequentialization technique is tailored for test generation, and available sequential testing techniques can be applied to the generated sequential programs without any modification.

5.6 Summary

In this chapter, we introduced a sequentialization technique based on the bounded-interference heuristic to verify the effectiveness the bounded-interference heuristic. Based on our sequentialization, a concurrent program is transformed to a sequential program such that the executions of the generated sequential program are equal to executions of the concurrent program (within a bounded number of interferences among threads). One advantage of our sequentialization technique is that (in contrast to the sequentialization techniques based on context bounding) state-of-the-art sequential testing techniques can be applied on the generated sequential programs without any modification to explore both input and interference spaces of concurrent programs. Furthermore, using sequential testing tools with coverage guarantees (like path coverage) would imply coverage guarantees (modulo the interference bound and computation limits) on the concurrent program after the testing process is finished. We implemented a prototype for multi-threaded C# programs. Our experiments showed that the bounded-interference heuristic is very effective in finding concurrency bugs.

Chapter 6

Bounded-Interference Concolic Testing of Concurrent Programs

Concolic testing [24, 73, 5, 84, 4] is a powerful technique in providing coverage guarantees for sequential programs.

a successful technique in testing sequential programs. Concolic testing assumes that programs are deterministic, i.e., they will take the *same execution path* when the same input values are given to them. Several advanced search algorithms over the input space (which is the only parameter for sequential programs) have been proposed and embedded in concolic testing, targeting different coverage criteria (e.g., path coverage, branch coverage, etc.). However, applying concolic testing to concurrent programs is very challenging. The behaviour of a concurrent program is influenced not only by input values but also by interleavings of execution of threads. Therefore, concolic execution of concurrent programs would result in a set of constraints that are closely tied to the specific schedule performed during program execution.

Data races have been used to leverage concolic testing to generate tests for concurrent programs [73, 70]. For example, jCUTE [73] is a concolic testing tool for multi-threaded Java programs that uses data races as a heuristic for interleaving exploration. It first executes the program with some random input values and observes an execution of the program under

some random thread scheduling. It identifies possible data races in the execution and repeatedly either generates new inputs (by keeping the schedule same as before) or generates a new schedule (by keeping the input values same as before) by re-ordering the events that form a data race. The main problem with this testing technique is that it can provide coverage guarantees only when the testing algorithm is terminated after considering all possible orderings of events involved in a data race. Note that this exploration space is often very large for real world programs such that the testing algorithm fails to terminate in a reasonable amount of time. Therefore, due to the data race heuristic, jCUTE is unable to quantify the partial work done (e.g., at the occasion of a timeout) as a meaningful coverage measure for the program.

In this chapter, we generalize concolic testing to concurrent programs, and hence we call it *concurrent concolic testing*, or (conc)^{2}olic testing in short [14]. However, we use the bounded-interference heuristic (proposed in Chapter 4) to guide the input/interleaving exploration in a way that we can quantify the effort spent on testing as a coverage measure. We introduce a new component in concolic testing, called *interference scenario exploration* component, that explores possible interference scenarios (within the interference bound) and for each of them generates a test (i.e., input values and a schedule) that realizes it (if possible). Using the interference scenario exploration component, we build a general testing framework where one can employ different strategies in exploring both input space and interference scenario space. We have implemented a search strategy that targets achieving maximal branch coverage for concurrent programs (time and space allowing) while considering a bounded amount of interferences among the threads.

(Conc)^{2}olic testing can theoretically guarantee *completeness* in the limit; i.e., if the testing algorithm runs for long enough without encountering memory issues, then, in the limit, we can cover every program branch or declare it unreachable. However, it can also provide coverage guarantees (modulo the maximum bound reached) at the occasion of timeouts or out-of-memory errors. Naturally, (conc)^{2}olic testing is limited by the same constraints that hold concolic testing back, namely, external function libraries or limitations of the SMT solvers for

undecidable logics.

We implemented the (conc)^{2}olic testing technique as a tool for testing multi-threaded C programs and used a set of benchmarks in concurrency research literature to demonstrate the practical efficiency of our technique in providing coverage and finding bugs in these benchmarks. We present the (conc)^{2}olic testing in this chapter in detail.

6.1 A Running Example

Figure 6.1 shows a buggy implementation of function `addAll` of a concurrent vector. We use this example to explain ideas and algorithms presented in this chapter. This example also nicely demonstrates why there is a need for systematic exploration of both input and interleaving spaces for testing concurrent programs.

Function `addAll` has two input parameters which are pointers to vector structures. It appends all elements of the second vector to the end of the first vector. Each `vector` has three fields: `data` which is an array holding vector elements, `size` which represents the size of `data`, and `cnt` which keeps track of the number of elements in `data`. Function `addAll` uses a lock `lk` to synchronize the calls to this function. It first checks whether there is enough space to insert all elements of `u->data` into `v->data`, i.e., `v->cnt` + `u->cnt` $\leq$ `v->size` (cf. line 4). If not, it increases the size of `v->data` accordingly. The invariant `v->size` $\geq$ `u->cnt` + `v->cnt` is stated as an assertion at line 8. Finally, it copies the elements and updates `v->cnt`. The bug in `addAll` corresponds to the fact that the value of `v->cnt` is being read (at line 2) outside the lock block and hence `v->cnt` can be changed by other threads before the lock block is executed, leading to an inconsistent state.

Imagine a concurrent program with two threads T and T', each of them calling `addAll` with `v` and `u` as arguments, where `v` is shared between the threads and `u` is an input of the program. Therefore, each individual field of `v` is treated as a shared variable and each individual field of `u` is treated as an input. Also, suppose that initially `v->cnt` is 10 and `v->size` is 20.

```
typedef struct {int cnt, int size, int* data} vector;
pthread_mutex lk;
1  void addAll(vector* v, vector* u) {
2    int numElem = v->cnt + u->cnt;
3    pthread_mutex_lock(&lk);
4    if(numElem > v->size) {
5      v->data = realloc(numElem * 2);
6      v->size = numElem * 2;
7    }
8    assert(v->size ≥ u->cnt + v->cnt);
9    ...  //copy data from u to v
10   v->cnt = v->cnt + u->cnt;
11   pthread_mutex_unlock(&lk);
12 }
```

Figure 6.1: A buggy implementation of function `addAll` of a concurrent vector.

Now, consider the situation where `u->cnt`=7 and the program is executed as follows:

(i) The first thread T executes line 2, reading 10 from `v->cnt`, 7 from `u->cnt` and storing value 17 in `numElem`.

(ii) The second thread T' is executed completely. It reads values 10 and 7 from `v->cnt` and `u->cnt`, respectively (at line 2) and assigns 17 to `numElem`. Then, it enters the lock block. Since `v->size` is greater than 17 it skips lines 5 and 6 and assigns 17 to the shared variable `v->cnt` before exiting the lock block.

(iii) Then, T continues execution: It skips lines 5 and 6 since (`numElem`=17) $<$ (`v->size`=20). However, when T gets to the assertion, `v->cnt` has value 17 written by T'. Therefore, (`v->size`=20) $<$ (17 + 7), and hence the assertion is violated.

This error is interesting because it requires a combination of a particular concurrent sched-

ule combined a with particular (relative) values for the input vectors to manifest. If the threads are executed sequentially back to back, nothing goes wrong. On the other hand, if we execute the same interleaving (as described above), but start with `u->cnt` having the value 3 (instead of 7), then nothing goes wrong again; the first thread assigns 13 to `numElem`, the second thread skips lines 5 and 6 and assigns 13 to `u->cnt`. Then, the first thread skips lines 5 and 6 since (`numElem`=13) $<$ (`v->size`=20). This means that triggering this concurrency bug does not solely depend on the schedule, nor does it solely depend on the chosen input values; it depends on finding the right combination of input values and a schedule. Any testing technique that does not explore the combination space systematically has the potential of missing on this bug.

6.2 Preliminaries

In this section, we introduce some notions from concolic testing adjusted to our application. Classical sequential concolic testing (discussed in Section 3.2.4) logs a set of path constraints over *input variables* during concolic execution which describes the conditions on the values of the inputs that have to be true to drive the execution of the program along the same path. However, doing the same for concolic execution of multi-threaded programs would result in a set of constraints that are closely tied to the specific schedule performed during program execution. To solve this problem, we proceed as follows: Instead of explicitly tracking scheduling decisions, we introduce symbolic variables which enable us to track the information flow between threads. More precisely, we introduce an additional symbolic variable each time a shared variable is read. Furthermore, for each shared variable write, we store the symbolic value (based on symbolic inputs and symbolic read variables). By doing so, we will be able to flexibly combine reads from and writes to shared variables and build a set of path constraints in a way which is not tied to a specific schedule but rather depends on a set of interferences among the threads.

In the following, we define the notion of *global symbolic traces*. Same as global traces introduced in Section 2.2.1, global symbolic traces does not contain any information about local

computations (i.e., accesses to local variables). However, for shared variable read and write events, instead of concrete values read or written during the execution, we record symbolic values. Furthermore, to be able to generate path constraints, global symbolic traces include branching decisions (that depend on shared variables or input values) made throughout the execution as well.

6.2.1 Global Symbolic Traces

Formally, a concurrent program consists of a set of threads $T = \{T_1, T_2, \ldots\}$, a set of input variables *IN*, a set of shared variables *SV*, a set of local variables *LV*, and a set of locks L that the threads manipulate. Let *SymbIN* be a set of symbolic input variables $\{i_0, i_1, \ldots\}$ and *SymbRV* be a set of symbolic shared read variables $\{r_0, r_1, \ldots\}$. Furthermore, let *Expr* represent the set of all expressions over *SymbIN* and *SymbRV*, and let *Pred(Expr)* represent the set of all predicates over *Expr*. Then, the set of actions Σ that a thread can perform is defined as:

$$\begin{aligned}\Sigma = \quad & \{\textit{rd}(x, r) \mid x \in \textit{SV}, r \in \textit{SymbRV}\} \cup \\ & \{\textit{wt}(x, val) \mid x \in \textit{SV}, val \in \textit{Expr}\} \cup \{\textit{tf}(T_i) \mid T_i \in T\} \cup \\ & \{\textit{ac}(l),\ \textit{rel}(l) \mid l \in L\} \cup \{\textit{br}(\psi) \mid \psi \in \textit{Pred(Expr)}\}\end{aligned}$$

Action $\textit{rd}(x, r)$ corresponds to reading symbolic value r from a shared variable x. Each time we observe a read from a shared variable during concolic execution, we introduce a new symbolic variable $r \in \textit{SymbRV}$ that is uniquely associated with that specific read. Action $\textit{wt}(x, val)$ corresponds to writing a symbolic value which is represented as an expression val to a shared variable x. To couple a read of x with a write to x, it is enough to connect the stored expression at write to the symbolic value of the read, i.e., $r = val$. Action $\textit{tf}(T_i)$ represents forking thread T_i. Actions $\textit{ac}(l)$ and $\textit{rel}(l)$ represent acquiring and releasing of lock l, respectively. Finally, action $\textit{br}(\psi)$ denotes a branch which requires predicate ψ to be true. We model assertions in a program by two branches, i.e., one branch for passing the assertion and one branch for violating the assertion.

We denote the execution of an action by a thread as an *event*. Formally, an event is a tuple $(T_i, a) \in T \times \Sigma$. Let EV denote the set of all possible events. During concolic execution, we observe a sequence of events, a so-called *global symbolic trace*:

Definition 6.2.1 (Global Symbolic Trace)**.** *A global symbolic trace is a finite string $\pi \in EV^*$. By $\pi[n]$, we denote the n^{th} event of π. Given a global symbolic trace π, $\pi|_{T_i}$ is the projection of π to events performed by T_i. A global symbolic trace π is thread-local, if $\pi = \pi|_{T_i}$ for some T_i.*

In this chapter, wherever we refer to symbolic traces (or shortly traces), we mean global symbolic traces. The inputs to the concolic execution engine (which is adapted to execute multi-threaded programs) are an input vector of the program and a schedule which exactly specifies the resulting program run:

Definition 6.2.2 (Program Run)**.** *Consider a* deterministic *concurrent program P. A (partial)* run *of P, represented by $R = P(\overline{in}, \sigma)$, is uniquely described by a valuation of the input variables IN (presented by $\overline{in}$) and a schedule σ. A* schedule *σ is defined by a sequence $(T_{i_1}, n_1)(T_{i_2}, n_2) \dots (T_{i_{m-1}}, n_{m-1})\ (T_{i_m}, -)$ where $T_{i_j} \in T$, for all $1 \leq j \leq m$, and $n_j > 0$, for $1 \leq j < m$, specifies the number of executed actions. A tuple $(T_{i_j}, -)$ represents the execution of thread T_{i_j} until T_{i_j} terminates. A program run $R = P(\overline{in}, \sigma)$ is* feasible *if P can be executed with input vector $\overline{in}$ and according to schedule σ. Each feasible program run R yields a symbolic trace $\pi(R)$.*

We assume that the program is instrumented in such a way that all program actions covered in EV are actually observed by $\pi(R)$.

Figure 6.2, on the left, shows a symbolic trace π obtained from the assertion violating execution of the program in Figure 6.1, discussed in Section 6.1. Note that concolic execution does not log any information about accesses to local variables. Internally, the concolic execution engine keeps track of the symbolic values of local variables and is therefore able to correctly update symbolic values written to shared variables.

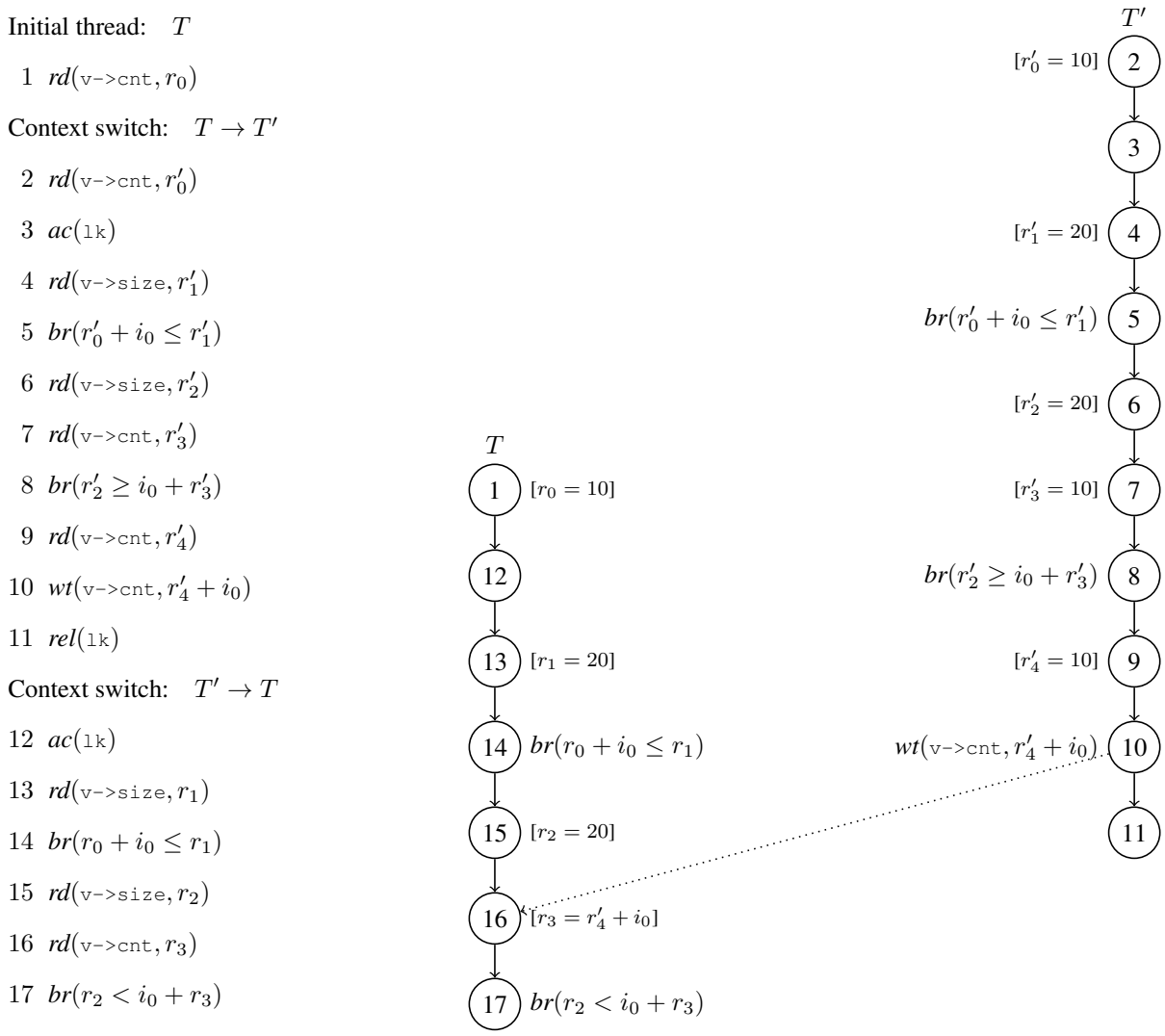


Figure 6.2: Symbolic trace π obtained from the assertion violating execution of the program presented in Figure 6.1 and its corresponding interference scenario $IS(\pi)$. i_0 represents the symbolic value of input `v->cnt`. r_0, r'_0, r'_3, and r'_4 read initial value 10, and r_1, r_2, r'_1, and r'_2 read initial value 20, and r_3 reads $r'_4 + i_0$.

6.3 Interference Scenarios

In this section, we formally define the concept of *interference scenarios*, its variations, and some applicable operations, which form the basis for (conc)^{2}olic testing. We also define two constraint systems to generate input values and schedules according to interference scenarios, respectively.

6.3.1 Concepts and Definitions

An interference occurs whenever a thread reads a value that is written by another thread. We introduce interference scenarios to describe a class of program executions under which certain interferences happen during concolic execution. Intuitively, an interference scenario is a set of thread-local symbolic traces extended with an interference relation between write and read events from different threads. We represent a set of interference scenarios in a data structure called *interference forest*. Formally, an interference forest is a finite labeled directed acyclic graph whose nodes represent events and whose edges express relations between events.

Definition 6.3.1 (Interference Forest)**.** *An* interference forest *is a tuple $I = (V, E, \ell)$ where V is a set of nodes, $\ell : V \to EV$ is a labeling function which assigns events to nodes. For $v \in V$ where $\ell(v) = (T_i, a)$, we also define Th$(v) = T_i$ and Ac$(v) = a$ to be the thread and the action of the corresponding event, respectively. The set of edges E is the disjoint union $E = E_L \dot\cup E_I$ of thread-local edges E_L and interference edges E_I. A thread-local edge (or, simply, a local edge) is an edge $(s,t) \in E_L$ where Th$(s) =$ Th(t). An interference edge $(s,t) \in E_I$ is an edge where Th$(s) \neq$ Th(t) and Ac$(s) = wt(x, val)$ and Ac$(t) = rd(x, r)$ for some x, val, and r. We require that E_I is an injective relation, i.e., each read is connected to at most one write by E_I. The thread-local edges can be naturally* partitioned *according to their threads, i.e., $E_L = E_{T_1} \dot\cup E_{T_2} \ldots \dot\cup E_{T_n}$. Each E_{T_i} induces a subforest G_{T_i} which consists of all nodes with Th$(v) = T_i$ and edges in E_{T_i}. We require that each G_{T_i} is a rooted tree. The number of interference edges $|E_I|$ is called the* degree *of the interference forest. Given*

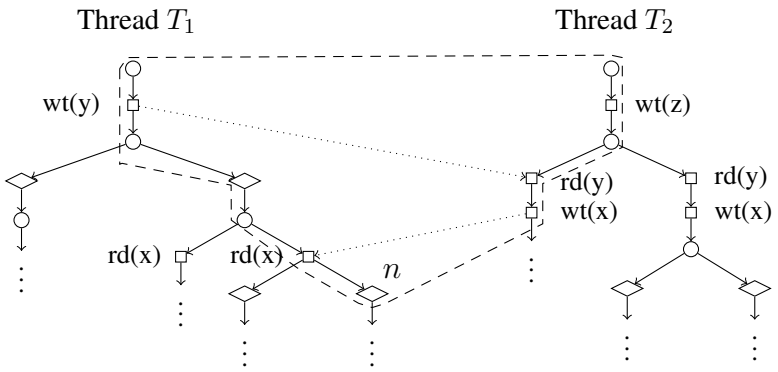


Figure 6.3: Example of an interference forest. Dashed lines enclose an interference scenario in the interference forest.

an interference forest J, $R_I(J)$ denotes the read nodes involved in the interference edges of J, i.e., $R_I(J) = \{n_r \mid \exists n_w.(n_w, n_r) \in E_I\}$.

Figure 6.3 shows an interference forest. The nodes labeled with read/write and branch actions are represented by squares and diamonds, respectively. Local edges are presented by arrows and interference edges are presented by dotted arrows. The left tree represents G_{T_1} and the right tree represents G_{T_2}. The degree of the interference forest is 2.

Definition 6.3.2 (Interference Scenario)**.** *An* interference scenario (IS) *is an interference forest where each G_{T_i} is a path.*

As mentioned at the beginning of this section, an interference forest is a compact representation for a set of interference scenarios.

Definition 6.3.3 (Causal Interference Scenario)**.** *Let $I = (V, E, \ell)$ be an interference forest. The transitive closure E^* of the edge relation E is called the causality relation of I. Given a node n, the* causal interference scenario (CIS) *of n is the subforest of I induced by the causal predecessors of n, i.e., by the node set $\{v \mid (v, n) \in E^*\}$. We denote it by $C = CIS(I, n)$ and call n the* sink *of C, i.e., sink(C) = n.*

Every causal interference scenario is itself an interference scenario. This is also the crucial property why interference forests serve as compact representations for sets of interference

scenarios. In Figure 6.3, the causal interference scenario of node n is the interference scenario enclosed by dashed lines.

Definition 6.3.4 (Isomorphic Interference Forests)**.** *Let $J = (V_J, E_J, \ell_J)$, $K = (V_K, E_K, \ell_K)$ be two interference forests. Let Rds_J and Rds_K be the symbolic variables corresponding to reads from shared variables in node labels of J and K, respectively. J and K are* isomorphic *if there exists a bijection $f : Rds_J \to Rds_K$, and a bijection $g : V_J \to V_K$ such that (i) $g(v_j) = v_k$ iff $\ell_I(v_k)$ is equal to $\ell_J(v_j)$ where each symbolic variable $r \in Rds_J$ is replaced by $f(r)$. (ii) $(u, v) \in E_J$ iff $(g(u), g(v)) \in E_K$.*

Isomorphism on interference forests is like isomorphism on labeled graphs while the isomorphic nodes have the same labels modulo symbolic variables corresponding to reads from shared variables. Construction of new interference scenarios from existing ones and merging interference scenarios into an interference forest are two central operations in (conc)^{2}olic technique. However, we cannot combine arbitrary interference forests/scenarios; they have to be compatible with each other:

Definition 6.3.5 (Compatible Interference Forests)**.** *Two interference forests I, J are* compatible *if there is an interference forest K and interference subforests I', J' of K such that I' and J' are interference forests themselves and I is isomorphic to I' and J is isomorphic to J'.*

Definition 6.3.5 also applies to compatible interference scenarios since each interference scenario is an interference forest by definition.

Remark 6.3.6. *Two compatible interference forests can be merged into an interference forest by naturally taking the minimal K, i.e., K only contains nodes and edges corresponding to I' and J'. Note that if interference scenarios I and J are not compatible, then there is at least one thread for which I and J describe different computations.*

Each symbolic trace π (obtained from a program execution) defines an interference scenario, denoted by $IS(\pi)$. Intuitively, each event represents a unique node in $IS(\pi)$ which is

labeled with that event. For each thread T_i, thread-local edges are added between the corresponding nodes according to the order in $\pi|_{T_i}$ (where $\pi|_{T_i} = \pi_{i,1}, \pi_{i,2}, .., \pi_{i,m}$ denotes the projection of events in π on thread T_i.) An interference edge is added for each node labeled with a read event if the last write event to the same shared variable before the read event in π is performed by another thread. More formally, $IS(\pi) = (V, E, \ell)$ is defined as:

- $V = \bigcup_{T_i}\{n_{i,j} \mid n_{i,j}$ is a unique node for event $\pi_{i,j}\}$,
- $\ell(n_{i,j}) = \pi_{i,j}$ for each node $n_{i,j}$,
- $E_{T_i} = \{(n_{i,k}, n_{i,k+1}) \mid 0 \leq k \leq m-1\}$, and
- $E_I = \{(n_{i,k}, n_{j,h}) \mid \mathit{Th}(n_{i,k}) \neq \mathit{Th}(n_{j,h}), \mathit{Ac}(n_{i,k}) = \mathit{wt}(x, val), \mathit{Ac}(n_{j,h}) = \mathit{rd}(x, r)$ for some x, val, r and $\pi_{i,k}$ is the last write to x in π before $\pi_{j,h}\}$.

Figure 6.2 shows the interference scenario for the symbolic trace obtained from the assertion violating execution of the program in Figure 6.1 discussed in Section 6.1.

Definition 6.3.7 (Realizable Interference Scenario)**.** *An interference scenario I is* realizable *in concurrent program P iff there is a feasible partial run R of P with $\pi = \pi(R)$ such that $IS(\pi)$ is isomorphic to I. We say R realizes I for a such feasible program run R.*

Realizable interference scenarios define equivalence classes on the set of program runs which represent the same flow of data among the threads. Note that interference scenarios are not monotonic wrt. realizability. Let I and I' be two interference scenarios where I is a subgraph of I'. Then, the realizability of I does not imply the realizability of I' and vice versa. We will discuss the reasons for this behaviour at the end of this section.

Interference scenarios specify partial program runs and therefore unanticipated behaviour can be observed:

Definition 6.3.8 (Unforeseen Interferences)**.** *Let I be a realizable interference scenario and R be a partial program run with $\pi = \pi(R)$ such that I is isomorphic to $IS(\pi)$. Let R' be a*

run that extends R, i.e., $\pi' = \pi(R')$ and π is a prefix of π'. Then, $IS(\pi')$ is a supergraph of $IS(\pi)$. More specifically, $IS(\pi')$ might contain some additional interferences. We refer to these interferences as interferences unforseen *by interference scenario I in run R'.*

6.3.2 Constraint Systems

Each interference scenario implies constraints on both data and temporal order of the events. Here, we describe these constraints in detail. In Section 6.4.3, we present a theorem (Theorem 6.4.3) that shows how these constraints can be used to check for the realizability of an interference scenario.

Data Constraints. Each interference scenario $I = (V, E, \ell)$ defines a data constraint $DC(I)$ as shown in Figure 6.4. Any solution to $DC(I)$ (if one exists), defines an input vector $\bar{i}$ for the concurrent program. The constraint $DC(I)$ consists of three parts: (i) DC_{branch}, (ii) $DC_{interference}$, and (iii) DC_{local}. The constraint DC_{branch} encodes all branch conditions occurring in I. The intuition behind this constraint is that the program execution should follow the control path in each thread represented by the respective branching conditions. $DC_{interference}$ relates each read from a shared variable, which should be interfered by a write from another thread, to the symbolic value of the corresponding write. Finally, DC_{local} relates each read from a shared variable, which should not be interfered by any write from other threads, to the most recent write to the same shared variable performed by the same thread. If there is no such write before the read, the symbolic value of the shared variable is constrained to the initial value of the variable. In this formula, let *intReads* represent all nodes v with $Ac(n) = rd(x, r)$ such that v is involved in an interference edge in E_I, and let *LocW* be a function that for each node v with $Ac(v) = rd(x, r)$ and $Th(v) = T_i$ returns a node u with $Ac(u) = wt(x, val)$ and $Th(u) = T_i$ in I such that u is the closest such node to v before v in G_{T_i}.

Temporal-Consistency Constraints. Each interference scenario I also defines a temporal consistency constraint $TC(I)$. Any solution to this constraint defines a schedule for the concurrent program. The constraints in $TC(I)$, as defined in Figure 6.4, are divided into the following

Data Constraints $DC(I)$:

$DC(I)$: $\quad DC_{branch}(V) \wedge DC_{interference}(I) \wedge DC_{local}(I)$

$DC_{branch}(V)$: $\quad \bigwedge_{\psi \in BR(V)} \psi$ where $BR(V) = \{\psi \mid v \in V, Ac(v) = br(\psi)\}$

$DC_{interference}(I)$: $\quad \bigwedge_{(v_{rd}, v_{wt}) \in E_I} DC_{match}(v_{rd}, v_{wt})$

$DC_{local}(I)$: $\quad \bigwedge_{v_{rd} \notin intReads} DC_{match}(v_{rd}, LocW(v_{rd}))$

$DC_{match}(v_{rd}, v_{wt})$: $\quad (r = val)$ for $Ac(v_{wt}) = wt(x, val), Ac(v_{rd}) = rd(x, r)$

Temporal-Consistency Constraints $TC(I)$:

$TC(I)$: $\quad \bigwedge_{T_i \in T} PO_{T_i} \wedge FC \wedge LC_1 \wedge LC_2 \wedge WRC_{interference} \wedge WRC_{local}$

PO_{T_i}: $\quad \bigwedge_{n_{i,j} \in G_{T_i},\ n_{i,j} \text{ is not a leaf}} (t_{n_{i,j}} < t_{n_{i,j+1}})$

FC: $\quad \bigwedge_{T_i \in T} (t_{n_{tf(T_i)}} < t_{n_{i,1}})$

LC_1: $\quad \bigwedge_{T_i \neq T_j} \bigwedge_{l \in L} \bigwedge_{\substack{[aq,rl] \in L_{T_i,l} \\ [aq',rl'] \in L_{T_j,l}}} \left(t_{rl} < t_{aq'} \vee\ t_{rl} < t_{aq}\right)$

LC_2: $\quad \bigwedge_{T_i \neq T_j} \bigwedge_{l \in L} \bigwedge_{\substack{aq \in NoRel_{T_i,l} \\ [aq',rl'] \in L_{T_j,l}}} (t_{rl'} < t_{aq})$

$WRC_{interference}$: $\quad \bigwedge_{(u,v) \in E_I} Coupled(v, u)$

WRC_{local}: $\quad \bigwedge_{v \notin intReads} Coupled(v, LocW(v))$

$Coupled(v, u)$: $\quad (t_u < t_v) \wedge \bigwedge_{n \in W_x \setminus \{u\}} ((t_n < t_u) \vee (t_v < t_n))$

Figure 6.4: Constraint systems $DC(I)$ and $TC(I)$ for an interference scenario $I = (V, E, \ell)$.

four categories: (i) thread-local program-order consistency (PO_{T_i}), (ii) thread-fork consistency (FC), (iii) lock consistency ($LC_1 \& LC_2$), and (iv) write-read consistency ($WRC_1 \& WRC_2$). For each node n in I, an integer variable t_n (timestamp) is considered to encode the *index* of the event of the node in a symbolic trace π. In the constraints in Figure 6.4, let $n_{i,j}$ represent the j^{th} node in G_{T_i}, and let $n_{tf(T_i)}$ represent the node n where $Ac(n) = tf(T_i)$. The constraints of $TC(I)$ are:

PO_{T_i}: Ensures that for thread T_i, the thread-local program order is respected in the schedule.

FC: Ensures that no thread can be scheduled before it is forked.

LC_1 & LC_2: Each lock acquire node *aq* with $Ac(aq) = ac(l)$ and $Th(aq) = T_i$ and its corresponding lock release node *rl* in T_i define a lock block, represented by $[aq, rl]$. Let $L_{T_i,l}$ be the set of lock blocks in thread T_i regarding lock l. LC_1 ensures that no two threads can be inside lock blocks of the same lock l, simultaneously. LC_2 ensures that the acquire of lock l by a thread that never releases it in I must occur after all releases of lock l in other threads. In this formula, $\mathit{NoRel}_{T_i,l}$ stands for lock acquire nodes in T_i with no corresponding lock release nodes.

$\mathit{WRC}_{\mathit{interference}}$ & $\mathit{WRC}_{\mathit{local}}$: Let W_x represent the set of all nodes u with $Ac(u) = wt(x, val)$, and *intReads* and *LocW* as defined before. For each read node v and write node u, the formula $\mathit{Coupled}(v, u)$ ensures that the read event of v is coupled with the write event of u in π by forcing all events that write to the corresponding variable to happen either before the event of u or after the event of v in π.

Non-Monotonicity of Realizability. We can now explain the non-monotonic behaviour of interference scenarios wrt. realizability that was mentioned in the discussion following Definition 6.3.7. Let I and I' be two interference scenarios where I is a subgraph of I'. Then, according to the data constraints, all constraints in $\mathit{DC}_{\mathit{branch}}(I)$ and $\mathit{DC}_{\mathit{interference}}(I)$ appear in $\mathit{DC}_{\mathit{branch}}(I')$ and $\mathit{DC}_{\mathit{interference}}(I')$, respectively. However, the constraints in $\mathit{DC}_{\mathit{local}}(I)$ and $\mathit{DC}_{\mathit{local}}(I')$ are incomparable. The same phenomenon exists in the temporal-consistency constraints, i.e., $\mathit{WRC}_{\mathit{local}}$ in I and I' are incomparable. This implies that, by extending an interference scenario, the resulting constraint systems do not change in a monotonic way.

6.4 (Conc)^{2}olic Testing

In this section, we first propose a general concolic testing framework for concurrent programs based on the notion of interference scenarios defined in the previous section. Then, we develop a concrete instance of this general framework by employing the bounded-interference heuristic in the search algorithm which is both sound and complete.

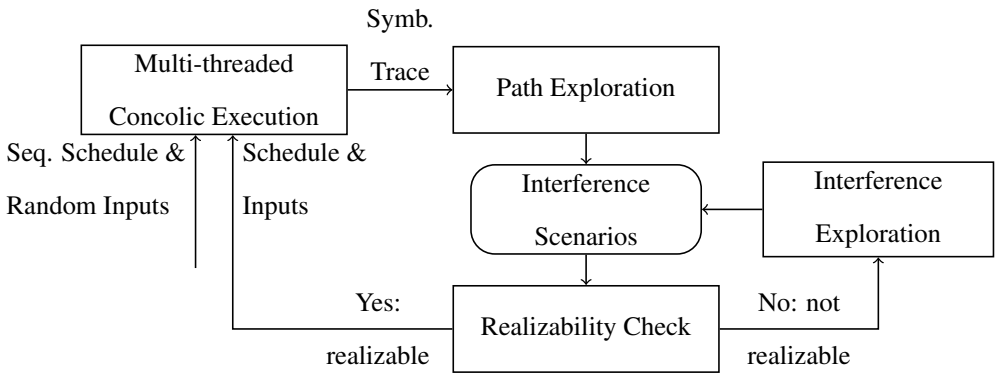


Figure 6.5: (Conc)^{2}olic testing framework.

6.4.1 General Framework

Figure 6.5 shows the (conc)^{2}olic testing framework. Similar to concolic testing (see Figure 3.3), (conc)^{2}olic testing has a concolic execution engine, a path exploration component and a realizability check component. However, (conc)^{2}olic testing has one more component, called *interference exploration* component, that explores the space of interference scenarios.

In (conc)^{2}olic testing, the execution engine is leveraged to execute a concurrent program with the provided input values based on a given schedule. Symbolic traces obtained from concolic execution are stored in an interference forest that keeps track of various interference scenarios that have already been explored. The path exploration component then, based on an already-seen interference scenario, aims to cover a previously uncovered part of the program (e.g., uncovered branches), by doing input exploration according to that interference scenario. Based on the interference scenario and a target branch defined by the path exploration, the realizability check component investigates whether there exist a set of input values and a *feasible* schedule such that the execution of the concurrent program with the inputs and based on the schedule results in covering the branch. If the answer is yes, the next round of concolic execution uses some input values and a schedule that realizes the interference scenario and covers the branch. In the case that the answer is no, the interference exploration component extends the interference scenario by introducing new interferences. In the following, we explain (conc)^{2}olic testing components in more detail.

Concolic Execution. There are two input parameters for the concolic execution engine in (conc)^{2}olic testing: (1) an input vector and (2) a schedule. The concolic execution engine executes the concurrent program with the given input vector and according to the given schedule. The program is instrumented such that, during the execution, all accesses to shared variables, synchronization events, and branching events are recorded to generate a global symbolic trace. This global symbolic trace contains all necessary information for the (conc)^{2}olic engine to make progress. However, it excludes any extra information, such as details of local computations of threads, that can safely be ignored in (conc)^{2}olic testing to gain scalability and efficiency.

Path Exploration. The role of the path exploration component is to explore the input space for a new set of input values that according to a previously-seen interference scenario covers a yet uncovered part of the program. As in concolic testing, it gets a symbolic trace and selects a branching event whose condition should be flipped to drive the execution towards an uncovered part of the program. The goal of path exploration is to use the exact set of interferences seen in a global symbolic trace and explore the input space based on that. Therefore, the path exploration component does not introduce any new interference scenario.

Realizability Checker. Getting an interference scenario with an uncovered target branch, the realizability checker determines if there is a set of input values and a feasible schedule such that the execution of the program with the input values and under the schedule realizes the given interference scenario and leads to covering the target branch. It generates the corresponding constraint systems (discussed in Section 6.3.2) for the interference scenario. There are two possibilities for such constraint systems:

(i) The combined constraint system has a solution. Then, any solution implies an input vector and a schedule which give rise to a program execution leading to covering the target branch with exactly the same set of interferences as defined in the given interference scenario. Therefore, we can formulate a new execution for the next round of the concolic execution module.

(ii) At least one of the constraint systems does not have a solution. This means that the interference scenario has to change. The current interference scenario is passed to the interference exploration module (described below), so that a new ones are produced based on it.

Interference Exploration. The interference exploration component produces new interference scenarios from previously explored interference scenarios, essentially by introducing a new interference. This is done by picking a read from the given interference scenario that is not interfered by other threads, and an appropriate write from the forest, and adding an interference from the write to the read to generate a new interference scenario. Note that the occurrence of the write event itself may be conditional on existence of other interferences. Therefore, to preserve soundness, all of those interferences should be included in the produced interference scenario as well.

Search Strategy. Having the above components, (conc)^{2}olic testing can exploit different search strategies and heuristics to explore the interference scenario space. We have developed an instance of it by employing the bounded-interference heuristic in the search strategy and targeting branch coverage. According to this search strategy, all interference scenarios with one interference are explored first, and then interference scenarios with two interferences are explored, and so on. A nice feature of this exploration strategy is that it is complete (Theorem 6.4.3) modulo the interference bound (and of course concolic testing limitations).

6.4.2 Testing Algorithm

Here, we present an algorithm that instantiates the (conc)^{2}olic testing framework by employing the bounded-interference heuristic in the search strategy and targeting branch coverage. Each assertion in the program can be modeled by two branches, one for passing the assertion and one for assertion violation. Therefore, our (conc)^{2}olic testing implicitly aims at finding assertion violations. We are specifically interested in interference scenarios related to nodes labeled with branch actions:

Definition 6.4.1 (Interference Scenario Candidate, ISC)**.** *Let n be a node with $Ac(n) = br(\psi)$, for some ψ. A causal interference scenario C is an* interference scenario candidate *for node n if $sink(C) = n$.*

Note that each ISC C with $sink(C) = n$ (if realizable) defines a set of partial program runs where $Ac(n)$ is the last action in the run. According to the bounded-interference heuristic, our algorithm enumerates all ISCs of degree i, and checks their realizability, before moving to ISCs of degree $(i + 1)$ for all $0 \leq i \leq k_{max} - 1$ where k_{max} is the interference bound.

Assumptions. For a concurrent program P, we assume that (i) individual threads in P are deterministic sequential programs. (ii) all threads are created by the main method of program P. Furthermore, to keep the exposition simple, we will make the following simplifying assumptions: (iii) There are no unforeseen interferences for an ISC C, i.e., each program run R' extending a partial run R, with $C = IS(R)$, results in an interference scenario $IS(R')$ which has exactly the same interferences as C. (iv) There are no locks in concurrent programs. Note that we state assumptions (iii) and (iv) for ease of presentation and our (conc)^{2}olic testing is not limited to settings where these assumptions are true; specially, all benchmark programs in Section 6.5.2 contain locks. We address removing these assumptions in Section 6.4.4.

Algorithm 3 shows our (conc)^{2}olic testing algorithm. Given a concurrent program P and a threshold k_{max} for the number of interferences, the algorithm explores ISCs of degree $\leq k_{max}$ with the aim of increasing branch coverage. For each such ISC, the algorithm tries to compute a corresponding test.

1-5: Data Structures. The algorithm utilizes three central data structures: (i) a global interference forest *forest* that stores all interference scenarios explored by concolic execution, (ii) a list of sets $W^0, \ldots, W^{k_{max}}$, where each W^k, for $0 \leq k \leq k_{max}$, serves as a worklist for ISCs of degree k, and (iii) a list of sets $UN^0, \ldots, UN^{k_{max}}$, where each UN^k, for $0 \leq k \leq k_{max}$, stores all processed but unrealizable ISCs of degree k. All these data structures are initially empty (cf. lines 1 to 5). During the execution of Algorithm 3, each generated ISC C of degree k is initially inserted into W^k. Later on, Algorithm 3 checks for the realizability of C and moves it

Algorithm 3: Test(program P, bound k_{max})

```
1  IForest forest ← ∅
2  ISC-Set W^0, ..., W^{k_max}, UN^0, ..., UN^{k_max}
3  for k = 0 to k_max do
4      W^k ← ∅
5      UN^k ← ∅
6  ī ← random inputs
7  foreach thread T_j do
8      π ← ConcolicExecution(P, (ī, (T_j, −)))
9      W^0 ← W^0 ∪ ExtractISCs(π)
10 for k = 0 to k_max do
11     while W^k ≠ ∅ do
12         pick and remove C from W^k
13         ISC-Set iscs ← ∅
14         (result, ī, σ) ← RealizabilityCheck(C)
15         if result ≠ realizable then
16             UN^k ← UN^k ∪ {C}
17             iscs ← ExploreISCs(C, write-nodes(forest))
18         else
19             π ← ConcolicExecution(P, (ī, σ))
20             W^k ← W^k ∪ ExtractISCs(π)
21             Wrts ← new-write-nodes(forest, π)
22             foreach C' ∈ UN^i, 0 ≤ i ≤ k − 1 do
23                 iscs ← iscs ∪ ExploreISCs(C', Wrts)
24         foreach C' ∈ iscs do
25             k' ← Degree(C')
26             if k' ≤ k_max then
27                 W^{k'} ← W^{k'} ∪ {C'}
```

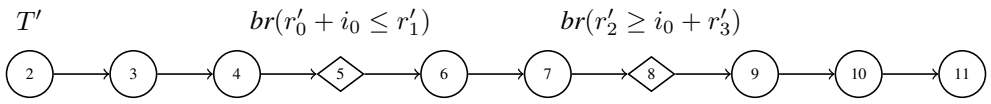
Algorithm 4: ExtractISCs (Symbolic Trace π) : ISC-Set

```
1 F ← addDanglingNodes(IS(π))
2 MergeInterferenceForests(forest, F)
3 ISC-Set iscs ← ∅
4 foreach dangling node n newly added to forest do
5     iscs ← iscs ∪ {CIS(forest, n)}
6 return iscs
```

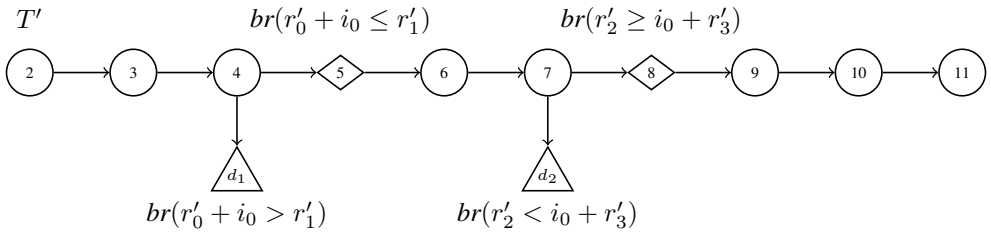
to UN^k, if it is not realizable, for further exploration.

6-9: Initial Path Exploration. We initialize W^0 by executing a test $(\bar{i}, (T_j, -))$ for each thread T_j (line 8), where $\bar{i}$ is a random input vector (we use the same $\bar{i}$ for each thread T_j) and the schedule $(T_j, -)$ allows only a sequential execution of thread T_j without any interruption from other threads. After concolic execution of thread T_j, program execution is aborted without executing any other thread, and a global symbolic trace π is returned. The global symbolic trace π, is passed to ExtractISCs (at line 9), which derives new ISCs for uncovered branches with the exactly the same set of interferences implied by π. Since π corresponds to a thread-local execution here, the degree of all generated ISCs is equal to 0. The ExtractISCs algorithm is described in the next paragraph. The returned ISCs are inserted into worklist W^0. After the initial path exploration phase, W^0 contains, for each thread in P, a set of ISCs for further exploration.

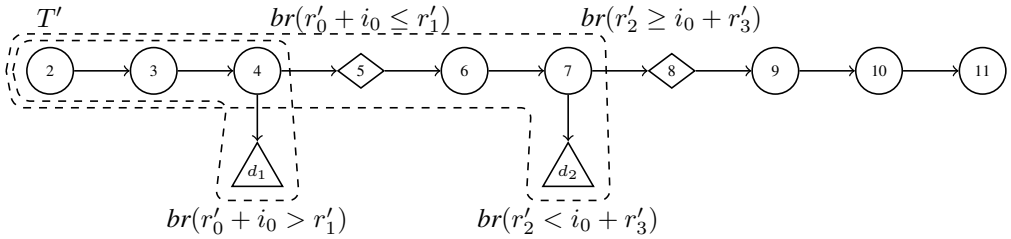
Algorithm ExtractISCs. ExtractISCs, shown in Algorithm 4, gets a global symbolic trace π as input. It first obtains an interference scenario according to π, i.e., $IS(\pi)$. For example, Figure 6.6a shows $IS(\pi_{T'})$ where $\pi_{T'}$ is the global symbolic trace returned by the initial sequential execution of thread T' (introduced in Figure 6.2). Then, the algorithm tries to generate new ISCs for uncovered branches according to $IS(\pi)$. Each branch node in $IS(\pi)$, has a corresponding dual branch node where its symbolic constraint is negated. ExtractISCs (at line 1), extends $IS(\pi)$ to an interference forest F by introducing for each branch, a dual branch node,



(a) Interference scenario $IS(\pi_{T'})$ for a symbolic trace $\pi_{T'}$ obtained by a sequential execution of thread T' (cf. 6.2).



(b) Interference scenario $IS(\pi_{T'})$ extended with dangling nodes d_1 and d_2.



(c) Interference scenario candidates $CIS(\mathit{forest}, d_1)$ and $CIS(\mathit{forest}, d_2)$.

Figure 6.6: An example showing initial path exploration for thread T' (cf. 6.2).

called a *dangling node*. Figure 6.6b shows the extension of the $IS(\pi_{T'})$ in Figure 6.6a by dangling nodes. The generated interference forest F is then merged into *forest* (cf. line 2) as described in Remark 6.3.6. Finally, for each dangling node which was not merged with an existing node, ExtractISCs creates an ISC (cf. lines 4 and 5). For example, $CIS(\mathit{forest}, d_1)$ and $CIS(\mathit{forest}, d_2)$ in Figure 6.6c are the ISCs generated by ExtractISCs from the interference forest in Figure 6.6b. Generated ISCs are returned to the main algorithm. Note that all of these ISCs will have the same interferences as $IS(\pi)$. Since *forest* is initially empty, during the initialization phase one ISC is generated for each dangling node in F.

10-27: Main Loop. The testing algorithm processes worklists $W^0, \ldots, W^{k_{max}}$ in ascending order. While processing W^k, each ISC $C \in W^k$ is removed from W^k and its realizability is

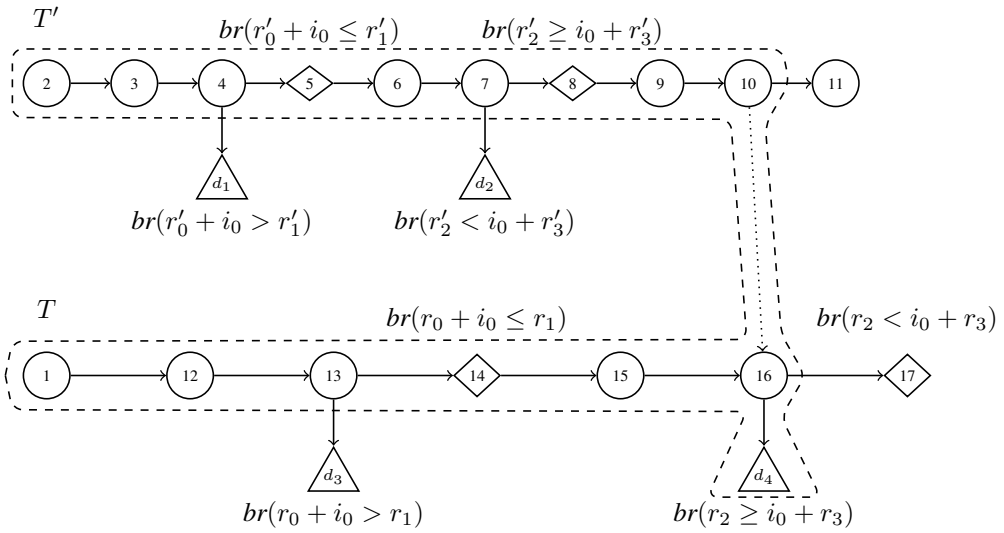


Figure 6.7: Interference scenario $IS(\pi)$ from Figure 6.2, extended by dangling nodes d_1, d_2, d_3, and d_4.

checked. Given an ISC *C*, RealizabilityCheck returns a triple $(\text{result}, \bar{i}, \sigma)$ where result indicates whether *C* is realizable or not. We discuss algorithm RealizabilityCheck in Section 6.4.3, in detail. If *C* is realizable, then $(\bar{i}, \sigma)$ forms a test that realizes *C*.

15-17: ISC Exploration. *C* is stored into UN^k for later processing if it is not realizable. Since the realizability of ISCs is not monotonic (as discussed in Section 6.3), *C* still has a chance to become realizable if some more interferences are introduced in it. Therefore, the algorithm collects all write nodes stored in *forest* (cf. line 17) and further calls ExploreISCs (Algorithm 5) to extend *C* to a set of ISCs for target branch *sink*(*C*) by introducing a new interference from a write in $Wrts$ to a read in *C*. Each of the generated ISCs has a degree $i > k$ and is added to W^i in lines 24 to 27. Since $i > k$, the newly generated ISCs will be processed after W^k is processed completely. We discuss ExploreISCs in detail later in this section.

19-20: Path Exploration. If *C* is realizable, then the program is concolically executed with input vector $\bar{i}$ and according to schedule σ (cf. line 19). The moment *sink*(*C*) is executed, the schedule σ enforces an exclusive execution of thread *Th*(*sink*(*C*)) without any interruption from other threads. The concolic execution returns a global symbolic trace π which is fed

Algorithm 5: ExploreISCs(ISC C, write nodes $Wrts$) : ISC-Set

```
1  ISC-Set iscs ← ∅
2  foreach n_r ∈ read-nodes(C) \ R_I(C) do
3      let Ac(n_r) = rd(x, r) for some x
4      foreach n_w ∈ Wrts do
5          if Ac(n_w) = wt(x, val) for some val then
6              if Th(n_w) ≠ Th(n_r) then
7                  I_w ← CIS(forest, n_w)
8                  if n_r ∉ R_I(I_w) and compatible(C, I_w) then
9                      C' ← merge(C, I_w)
10                     C'' ← extend C' by interference (n_w, n_r)
11                     //uncomment the following code for supporting locks
12                     //C'' ← ExtendToLockFreePoints(C'', forest)
13                     iscs ← iscs ∪ {C''}
14 return iscs
```

to ExtractISCs to update *forest* and derive ISCs from π similar to the $k = 0$ case described earlier. Figure 6.7 shows an example where $k > 0$. There, the loosely dashed lines enclose the interference scenario candidate $CIS(forest, d_4)$. All generated ISCs will be added to W^k.

21-23: ISC Re-Exploration. When *forest* is updated with $IS(\pi)$ during the path exploration, some write nodes might be added to the *forest*. For each of these write nodes, all previously unrealizable ISCs have to be reconsidered and extended (if possible) by an interference from the new write node to a read node in these ISCs. This happens at lines 21 to 23; each previously unrealizable ISC C with degree smaller than k is re-explored by introducing an interference corresponding to the newly observed write (if possible). All of these writes requires exactly k interferences to happen since they occur after $sink(C)$ is covered. Therefore, the ISCs generated at line 23 have a degree greater than k and are added to the according worklists at lines 24–27.

Algorithm ExploreISCs. Algorithm ExploreISCs explores ISCs by extending existing ISCs with new interferences. Let n_r be a read node in a given ISC C. Let n_w be a write node in *forest* and let I_w be the causal interference scenario of n_w, i.e., $I_w = CIS(\mathit{forest}, n_w)$. To create an ISC C'' which extends C by the interference (n_w, n_r), the algorithm checks the following conditions:

1. n_r and n_w are in different threads, i.e., $\mathit{Th}(n_r) \neq \mathit{Th}(n_w)$.
2. n_r and n_w correspond to the same shared variable, i.e., if $\mathit{Ac}(n_r) = \mathit{rd}(x, r)$, for some symbolic variable r, then $\mathit{Ac}(n_w)$ is of the form $\mathit{wt}(x, val)$, for some expression val.
3. n_r is not involved in any interference in C or I_w, i.e., $n_r \notin R_I(C)$ and $n_r \notin R_I(I_w)$ (cf. Section 6.3).
4. C and I_w are compatible (cf. Definition 6.3.5).

If all conditions are satisfied then C and I_w are merged as described in Remark 6.3.6 and form an ISC C'. Then, C' is extended to ISC C'' by introducing the interference edge (n_w, n_r). Algorithm 5 collects all generated ISCs and finally returns them at line 14. Note that each generated ISC C'' has the same sink as C, i.e., $\mathit{sink}(C'') = \mathit{sink}(C)$, and has at least one more interference than C, i.e., $\mathsf{Degree}(C'') \geq \mathsf{Degree}(C) + 1$. The degree of C'' might increase by more than one interference, because I_w might contain interferences which are not present in C, but, due to the merge, they show up in C' and C'' as well.

6.4.3 Soundness and Completeness

In Section 6.3.2, we discussed data constraints $DC(I)$ and temporal constraints $TC(I)$ corresponding to an interference scenario I. In the following, we first show that these constraints can be used to check the realizability of I which guarantees soundness. Then, prove that Algorithm 3 is complete.

The soundness of our (conc)^{2}olic testing is proved by the following lemma:

Lemma 6.4.2. *Assume that TC(C) and DC(C) are satisfiable for an ISC C generated by Algorithm 3. Let $\bar{i}$ be a model for DC(C) and σ be a schedule obtained according to the values of timestamps (sorted in ascending order) from a model of TC(C). Then, $R = P(\bar{i}, \sigma)$ is a feasible run of program P and $IS(R)$ is isomorphic to C.*

The proof of this lemma is provided in Appendix B.

Theorem 6.4.3. *Let C be an ISC generated by Algorithm 3. C is realizable if and only if DC(C) and TC(C) are satisfiable.*

Proof. $\Rightarrow$: First, we assume that C is realizable, i.e., there exists a program run $R = P(\bar{i}, \sigma)$ such that $IS(R)$ is isomorphic to C. In this case $\bar{i}$ and σ are models for DC(C) and TC(C), respectively. Therefore, both DC(C) and TC(C) are satisfiable.

$\Leftarrow$: Now, we assume that if for an ISC C generated by Algorithm 3, DC(C) and TC(C) are satisfiable. Let $\bar{i}$ be a model for DC(C) and σ be a schedule obtained from a model of TC(C). According to Lemma 6.4.2, $R = P(\bar{i}, \sigma)$ is a feasible run of program P and $IS(R)$ is isomorphic to C. Therefore, C is realizable. □

Algorithm RealizabilityCheck. Given an ISC C with *sink*(C) $= n$, the realizability of C is checked by determining whether DC(C) and TC(C) are both satisfiable. Assume that C is realizable, $\bar{i}$ is a model for DC(C) and σ' is a schedule obtained according to the values of timestamps (sorted in ascending order) from a model of TC(C). Then, the RealizabilityCheck algorithm (called in Algorithm 3 at line 14) returns a triple $(\text{result}, \bar{i}, \sigma)$ where result determines that C is realizable and $\sigma = \sigma'(Th(n), -)$ which forces the sequential execution of thread $Th(n)$ after σ'.

Theorem 6.4.4 (soundness)**.** *Each program test run $R = P(\bar{i}, \sigma)$ generated by Algorithm 3 to realize an ISC C is feasible and $IS(R)$ is isomorphic to C.*

Proof. A program test run $R = P(\bar{i}, \sigma)$ is generated whenever an ISC C is found to be realizable. In this case, $\bar{i}$ is a solution to DC(C) and σ is a schedule according a model of TC(C). According to Lemma 6.4.2, R is feasible and $IS(R)$ is isomorphic to C. □

To state the completeness theorem, we first define the notion of *k-coverable* branches:

Definition 6.4.5 (k-coverable branch)**.** *A branch of concurrent program P is k-coverable if it requires k interferences to be covered and k is minimal.*

The completeness of our (conc)^{2}olic testing is proved by the following lemma:

Lemma 6.4.6. *For each k-coverable branch br where $k \leq k_{max}$, either $k = 0$ and br is covered by the initial random test (at line 8) or there exists a realizable ISC C in W^k (sink(C) might be different from br) whose generated test covers br. This implies that all writes that require k interferences to happen are added to the interference forest while processing W^k.*

The proof of this lemma is provided in Appendix C.

Theorem 6.4.7 (Completeness)**.** *Given a concurrent program P and a bound k_{max} on the number of interferences, Algorithm 3 covers all k-coverable branches of P where $0 \leq k \leq k_{max}$ (modulo concolic execution limits).*

The theorem is implied trivially by Lemma 6.4.6. Like all completeness results in concolic testing our completeness theorem relies on several idealizing assumptions. The theorem states that for deterministic programs without non-linear arithmetics and calls to external library functions, our (conc)^{2}olic testing algorithm covers all branches of P that require at most k_{max} many interferences to be covered. In practice, concolic execution falls back upon concrete values observed during execution to handle non-linear computations or calls to external library functions, for which no good symbolic representation is available.

6.4.4 Relaxing Assumptions

Dealing with Unforeseen Interferences. In order to drop assumption (iii) stated at the beginning of Section 6.4.2 about unforseen interferences, we need to make the following changes: (1) The concolic execution engine stops as soon as an unforeseen interference is observed and returns a global symbolic trace π that ends with the read event of the unforeseen interference.

(2) The ExtractISCs algorithm is extended as follows: When building forest F at line 1, a distinguished *dangling node* is added which is labeled with the read event of the unforeseen interference. However, the unforeseen interference is not added to F. As an effect, ExtractISCs algorithm will then create a causal interference scenario for this special dangling node (at line 5). Consequently, Test algorithm will then try to realize this interference scenario, first without introducing any interference. If this is not possible then it will introduce some interferences later. Note that the notion of CIS should be extended to allow sink nodes labeled with read events. Since our algorithms never make use of the fact that the sink of a CIS is a branch node, the overall testing algorithm is not changed.

Dealing with Locks. At the beginning of Section 6.4, we assumed that programs do not contain locks. Here, we present the issues that locks in programs introduce and discuss how the testing algorithm can be changed slightly to handle programs with locks. Consider an ISC C with $sink(C) = n$. It might be the case that for a thread $T_i \neq Th(sink(C))$, the last node in G_{T_i} is labeled with a write event (interfering with a read node in $G_{Th(sink(C))}$) that happened while T_i was holding some locks. This situation may cause the following problems: (i) C might never become realizable, e.g., if n is also protected by the same locks, then T_i does not have any chance to release the locks for $Th(sink(C))$. (ii) C might be realizable but the test generated for C may lead to a deadlock, e.g., thread $Th(sink(C))$ acquires any of these locks later.

To solve the problems, whenever we create a new ISC C, we extend all thread-local subscenarios of C according to *forest*, except for thread $Th(sink(C))$, such that for each thread T_i the last node in G_{T_i} according to C is not protected by any lock. This change to the testing algorithm can be done by uncommenting the commented line in ExploreISCs algorithm; assume that for an ISC C algorithm ExtendToLockFreePoints creates the extension. As an example consider the ISC shown in Figure 6.7. There, T' holds a lock at node 10. Therefore, the ISC is extended to include node 11 where T' releases the lock. We assume that this extension is always unique, i.e., each lock acquire operation corresponds to exactly one lock release operation in the code, where both operations are in the same block of code under the same branches.

Note that this assumption is not unrealistic since many widely-used programming languages (e.g., Java, C#, etc.) use lock block instructions to handle locks (i.e., the lock is acquired at the beginning of the lock block and is released at the end of the lock block) that inherently satisfy the above assumption.

6.4.5 Optimizations

Unsat-Core Guidance. In ExploreISCs algorithm, ISC exploration is performed by adding new interferences to unrealizable ISCs. For an unrealizable ISC, it might be the case that no extension of it by interferences will ever get realizable. From the unsatisfying core of the constraint systems (defined in Section 6.3.2), we can identify such situations. Let $C = (V, E, \ell)$ be an ISC. Data constraint $DC(C)$ is then equal to $DC_{branch}(V) \wedge DC_{interference}(C) \wedge DC_{local}(C)$. Extending C to a new interference scenario C' by adding an interference to C removes some predicates in $DC_{local}(C)$ from $DC(C')$ but the predicates in $DC_{branch}(V)$ and $DC_{interference}(C)$ remain as part of $DC(C')$. Therefore, if the unsatisfying core of $DC(C)$ does not involve predicates from $DC_{local}(C)$, we can conclude that $DC(C')$ or any other extension of C will not be realizable as well and, therefore, we can exclude C from further exploration. Analogously, if $TC(C)$ is not satisfiable and no constraints from WRC_{local} are involved in the unsatisfying core then, again, we can conclude that C will not become realizable by adding new interferences and we can exclude C from further exploration. Furthermore, in both cases, the unsat core can be used to guide the exploration by introducing an interference for a so far local read whose constraints are involved in the unsat core.

Duplication Freedom. The ExploreISCs algorithm allows multiple instantiations of the same ISC. For example, suppose that an ISC C becomes realizable by introducing interferences for two reads. The algorithm can first select any of these reads and generate two ISCs in which one of these reads is interfered. Then, in the future, these two ISCs can be extended such that the other read is also interfered, generating two instances of the same ISC. To avoid duplication of ISCs, we use a caching mechanism. In this way, an ISC will be processed only if it is not

already in the cache.

Prioritized Exploration. While processing each worklist W^k, we can choose to prioritize the ISCs in W^k. For example, in our implementation, we assign higher priorities to ISCs which would cover some yet uncovered part of program code (in case they are realizable). Based on this exploration strategy, the ExploreISCs algorithm (at line 11) first processes ISCs with higher priorities.

6.5 Evaluation

We implemented our (conc)^{2}olic testing technique in a tool called CONCREST. We built CONCREST on top of CREST [4] which is a concolic testing tool for sequential C programs. To evaluate CONCREST, we subjected it to a benchmark suite of multi-threaded C programs. In the following, we briefly discuss the implementation and then present our experimental results.

6.5.1 Implementation

In Section 6.4.1, we discussed how a traditional concolic testing technique can be leveraged to (conc)^{2}olic testing. Figure 6.5 shows the high-level components of (conc)^{2}olic testing. We implemented (conc)^{2}olic testing by extending a sequential concolic testing tool CREST [4] as follows: (i) Concolic execution engine is changed such that in addition to an input vector, it also gets a schedule, executes the program with the given input vector according to the schedule and generates a global symbolic trace. (ii) The path exploration component is adapted to Algorithm 4. (iii) An interference scenario exploration component is added which realizes Algorithm 5. (iv) The constraint system generated by the realizability check component is lifted as discussed in Section 6.3.2. (v) The search algorithm is changed based on the bounded-interference heuristic according to Algorithm 3.

Program	♯Thrd	♯Inputs	♯Br (total)	♯Br k=0/1/2/3/4/	Max k reached (reason)	♯Br k=0 → Max	Bug found (k)	♯ ISC (total)	time (total)
bluetooth	3	2	24	14/8/2	2 (Full Cov.)	14→24	yes(2)	282	1
sor	3	-	48	37/8/0/0/3	4 (Full Cov.)	37→48	yes(3)	145	1
ctrace1	3	-	94	54/3	5 (Max Cov.)	54→57	yes(1)	28	1
apache2	3	3	72	41/0/1	11 (Max Cov.)	41→42	yes(2)	392	1
splay	3	-	112	46/14/4	15 (Max Cov.)	46→64	no	3501	6
apache1	3	3	48	35/3	11 (Max Cov.)	35→38	yes(1)	22150	15
aget	3	-	88	56/0/1	21 (Max Cov.)	56→57	yes(2)	23197	170
rbTree	3	-	146	67/22/4/2	24 (Max Cov.)	67→95	no	77037	296
pfscan	3	2	130	92/0/0/0/1	4 (Timeout)	92→93	yes(4)	3012548	7200
ctrace2	3	-	128	75/5	2 (Timeout)	76→81	yes(1)	315639	7200
art2	3	2	8	7/1	1 (Full Cov.)	7→8	yes(1)	80	0.3
art3	4	3	12	10/1/1	2 (Full Cov.)	10→12	yes(2)	17942	21.8
art4	5	4	16	13/1/1/1	3 (Full Cov.)	13→16	yes(3)	2842066	197.1
art5	6	5	20	16/1/1/1/1	4 (Full Cov.)	16→20	yes(4)	10851573	741.1

Table 6.1: Experimental results for (conc)^{2}olic testing according to bounded-interference heuristic. **♯Br**: number of *static* branches, i.e., number of basic code blocks. **k**: number of interferences. **Full Cov.**: all branches are covered. **Max Cov.**: all possible interference scenario candidates are explored.

6.5.2 Experiments

Benchmarks: `bluetooth` is a simplified version of the Bluetooth driver from [62]. `sor` is from Java Grande multi-threaded benchmarks (which we translated to C). `ctrace1` and `ctrace2` are two test drivers for the `ctrace` library. `apache1` and `Apache2` are test drivers for APACHE FTP server from BugBench [46]. `splay` and `rbTree` are test drivers for a C library which implements several types of trees. `aget` is a multi-threaded download accelerator. `pfscan` is a multi-threaded file scanning program. Finally, `art` is an example designed by us to evaluate the scalability of our (conc)^{2}olic testing according to the number of threads. It has the property that a new assertion can be violated every time we increase the number of threads by one. In this example, there is a shared variable x among the threads. Each thread has an

integer input i such that $0 \leq i \leq 5$, and performs $x = x + 10^i$ for 9 times in a loop. There is also an assertion in the loop checking that x does not have a specific value.

The experiments were performed with $k_{max} = 100$ (at most 100 interferences) and a time-out of 2 hours. In Table 6.1, we report the number of threads and inputs, the total number of *static* branches in the benchmarks, the number of static branches covered by having 0/1/2/3/4/etc. interferences, the maximum bound reached for the number of interferences (and the reason why it did not go beyond it), branch coverage improvement over sequential testing (i.e., 0 interference), if any bug was found (and the number of interferences required to expose the bug), the total number of explored ISCs, and the time spent on testing, respectively.

Observations: The experiments show that CONCREST is effective in increasing branch coverage. For some of the benchmarks, a substantial number of branches were not sequentially coverable and were only covered after interferences were introduced, e.g., for `rbTree`, the number of covered branches increases from 67 in sequential testing to 95 in (conc)^{2}olic testing.

From the maximum bound reached in each benchmark for the number of interferences, we can see that although we set the maximum number of interferences to be 100, the actual bound explored by CONCREST is much smaller. This is because in most cases (with the exception of 2 timeout cases), we either achieved maximum branch coverage or explored all possible ISCs (i.e., no more branches are coverable). In the lack of a bug found, reaching the maximum coverage provides guarantees to the tester that, e.g., no assertions in the code can be violated.

There are cases where maximum branch coverage is achieved, but the number does not coincide with the total number of static branches. We found that the remaining branches were either not coverable by the test driver, or were branches on local variables, related to sanity checks on the system execution. Such sanity checks include checks on system call executions, such as whether a file-open operation using `fopen` succeeded or not. Since our test execution was not providing any mock environment, such as for the file system, these sanity checks could not fail.

Row	U	P	D	Assertion Violation Found (time (s))	Max k Reached	Total Time (s)
1	+	+	+	yes (4554)	4	7200 (timeout)
2	-	+	+	no	2	7200 (timeout)
3	+	-	+	yes (6701)	4	7200 (timeout)
4	+	+	-	no	3	7200 (timeout)
5	-	-	+	no	2	7200 (timeout)
6	-	+	-	no	2	out of memory
7	+	-	-	no	3	7200 (timeout)
8	-	-	-	no	2	out of memory

Table 6.2: Optimization effects on `pfscan` benchmark. **U** = unsat core guidance, **P** = prioritized exploration, **D** = duplication freedom. Symbols + and - represent optimizations being on and off, respectively.

Another observation is that CONCREST is very effective at finding bugs; all of the known bugs in the benchmarks were discovered by CONCREST. All of the bugs found in our benchmarks were revealed by covering a branch that could not be covered by sequential testing. Moreover, as the table shows, all bugs were discovered under a relatively small number of interferences (maximum 4). This implies that concurrency bugs are not very complex according to the number of interferences they require to be revealed.

Furthermore, we can see that the time spent by CONCREST is very reasonable. In all of the benchmarks (except two), we could get full coverage or maximum coverage in a few minutes. The interference spaces of `pfscan` and `ctrace2` were so huge that the interference bound could not go beyond 4 and 1 (within the time limit of 2 hours), respectively. However, CONCREST was able to find the bugs in these benchmarks within the time limit.

Effect of Optimizations: Table 6.2 presents the effects of the optimizations discussed in Section 6.4.5 for `pfscan` (as an example). The bug in `pfscan` corresponds to an assertion violation which is discovered at $k = 4$. For each configuration of the optimizations, we report whether the assertion violation is found (and the time spent on testing until it is found), the maximum number of interferences explored, and the total time spent for testing.

	(conc)^{2}olic testing			MTA	
Program	♯ Br. covered by k>0	Time to find the bug (s)	timeout?	♯ Br. covered by MTA	Time to find the bug (s)
apache1	3	1	no	2	32
apache2	1	1	no	1	81
aget	1	3	no	1	180
splay	18	no bug	no	8	no bug
ctrace1	3	1	no	3	341
ctrace2	5	757	yes	14	447

Table 6.3: Comparing (conc)^{2}olic testing with MTA

We can observe that when there is no optimization enabled, CONCREST runs out of memory with $k = 2$. The efficiency of unsat-core guidance is clear because without this optimization k cannot go higher than 2. In fact, to move to $k = 4$ and catch the assertion violations, both unsat-core guidance and duplication-freedom optimizations have to be enabled. The effect of prioritized exploration can be observed by comparing rows 1 and 3: when prioritization is enabled the assertion violation is found earlier. Therefore, all of the optimizations are effective in reducing the exploration space (and the time spent on exploration accordingly) to catch program bugs.

Comparison with MTA: The goal of multi-trace analysis (discussed in Chapter 3) is to increase branch coverage in concurrent programs. (Conc)^{2}olic testing also generates tests targeting branch coverage. However, it has another goal as well, which is to provide coverage guarantees (modulo the explored interference bound). We performed some experiments to compare these testing techniques against each other. In Chapter 3, we discussed how MTA is implemented on top of FUSION. As mentioned in that chapter, FUSION performs a simplification process on program approximations (i.e., set of program runs) by omitting branches which depend only on local variables or relate to sanity checks on the system execution and does not include them in the total number of branches. Therefore, a side by side comparison of the percentage of branch coverage in both techniques is not appropriate. To be fair, we compare the number of branches covered by MTA (after sequential testing of individual threads) and

(conc)^{2}olic testing (with $k > 0$ interferences). Furthermore, we compare the time spent by each of these techniques to catch the bugs in the benchmarks.

In Table 6.3, we consider the benchmarks used to evaluate both testing techniques (i.e., benchmarks common in Tables 6.1 and 3.1). For each benchmark, we report the number of branches covered after sequential testing and the time spent until the bug was found by each technique. We consider a timeout of 2 hours for (conc)^{2}olic testing (as in Table 6.1) and also report if the (conc)^{2}olic testing of the benchmarks reaches the timeout.

Comparing the number of covered branches, we can see that except in `ctrace2` (where (conc)^{2}olic testing reaches the timeout), (conc)^{2}olic testing performs better than MTA. This is because (conc)^{2}olic testing guarantees to cover each branch that can be covered by some tests (modulo the interference bound) where as MTA employs heuristics in selecting the interloper segments and exploring the input and interleaving spaces that do not provide coverage guarantees.

Another observation is that for benchmarks where (conc)^{2}olic testing does not reach the timeout, MTA requires more time to find the bug. This is because the analysis in (conc)^{2}olic testing is performed at global computation level (i.e., it is based on accesses to shared variables and synchronization events), while MTA considers and encodes all computation (local as well as global) in the runs which naturally increases the size of the test generation problem and correspondingly affects the computation time.

From the above observations, one might conclude that (conc)^{2}olic testing is better than MTA in any ways. However, that is not true. In case of `ctrace2`, where (conc)^{2}olic testing reached the timeout, MTA performs better than (conc)^{2}olic testing regarding both branch coverage and bug finding time. The interference scenario space of `ctrace2` is so huge that (conc)^{2}olic testing could not get to $k = 2$ in 2 hours. In this case, MTA required less time to find the bug and had a chance to cover more branches. These branches are covered under interference scenarios that (conc)^{2}olic testing did not get the chance to explore them. This suggests that for programs with large interference scenario spaces, MTA might actually perform better

than (conc)^{2}olic testing regarding bug finding. Therefore, both techniques have their own right of existence.

Conclusion: Our experiments showed that (conc)^{2}olic testing is effective in increasing branch coverage in concurrent programs. In fact for many of bur benchmarks, it was able to provide maximum branch coverage that provides bug finding guarantees for the tester, e.g., no more assertion in the code can be violated. Furthermore, (conc)^{2}olic testing was able to find a large number of bugs in our benchmarks. All of the bugs in our benchmarks were found by allowing only a few number of interferences among threads which shows the effectiveness of the bounded-interference heuristic in bug finding. We also showed that all of the optimizations that we proposed for reducing the interference scenario exploration space are necessary for scalability of (conc)^{2}olic testing. Furthermore, we compared (conc)^{2}olic testing with MTA. Our experiments showed that none of them outperforms the other and both techniques have their own right of existence. (Conc)^{2}olic testing performs better than MTA for programs with manageable interference scenario spaces as it can provide maximum coverage guarantees for them. However, for programs with huge interference scenario spaces, MTA may perform better in bug finding as it might be able to try interference scenarios that the (conc)^{2}olic testing does not have a chance to try in a reasonable amount of time.

6.6 Related Work

Concolic testing of multi-threaded programs has been introduced by Sen et al. [73, 72, 70] and realized in a tool for testing concurrent java programs, named jCUTE. There are several differences between the technique proposed by Sen et al. and (conc)^{2}olic testing: Their technique uses data races as a heuristic to limit the interleaving space, i.e., interleaving exploration is done based on the data races found in previous executions by delaying the execution of the threads at the data race points to get schedules in which the order of the execution of the events involved in a data race is flipped. (Conc)^{2}olic testing uses the bounded-interference heuristic

to reduce the exploration space. This algorithm is proved to be complete in [70]. However, in contrast to (conc)^{2}olic testing that provides coverage guarantees modulo the maximum bound reached, jCUTE cannot provide coverage guarantees on partial work done (e.g., at the occasion of hitting time or memory limitations). This is due to the use of the data race heuristic that does not quantify the partial work done as a meaningful coverage measure for the program.

Our notion of interference-based search is related to work by Wang et al. [80, 79], where they use concurrent trace programs as an under-approximation of the programs and explore the interference scenario space to find bugs in program approximations. The proposed method in [80] utilizes both over- and under-approximations of interferences among the threads in concurrent trace programs to capture suitable interferences for finding assertion violations. In [79], a two-staged analysis is proposed which separates intra- and inter-thread reasoning. The first stage uses sequential program semantics to obtain a precise summary of each thread in terms of the global accesses made by the thread. The second stage performs inter-thread reasoning by composing these thread-modular summaries using the notion of sequential consistency to find assertion violations. However, there are several differences between these techniques and (conc)^{2}olic testing: They work on program approximations as opposed to programs. Furthermore, they target discovering assertion violations (by performing a symbolic analysis) in concurrent trace programs as opposed to generating tests for exploring different program behaviours. Moreover, their analysis is based on symbolic traces, which include all computation (local as well as global) and synchronization in program executions. The analysis in (conc)^{2}olic testing uses *global* symbolic traces, which ignores local computation, and therefore is much more scalable.

6.7 Summary

In this chapter, we adapted a sequential concolic testing technique, based on the bounded-interference heuristic, to generate tests for concurrent programs. We introduced a new com-

ponent in concolic testing, called interference scenario exploration component, that explores possible interference scenarios (within the interference bound). Using the interference scenario exploration component, we built a general testing framework where one can employ different strategies in exploring both input and interference scenario spaces. We have developed a search strategy that targets providing maximum branch coverage by incrementally increasing the interference bound. Therefore, it is able to provide coverage guarantees (modulo the interference bound and concolic execution limitations) after the testing process is finished or when a time or computation limit is reached. We proved that our testing technique is both sound and complete. We implemented the testing technique by leveraging the concolic testing tool CREST to test multi-threaded C programs and performed some experiments that show the effectiveness of (conc)^{2}olic testing in increasing branch coverage and finding concurrency bugs.

Chapter 7

Conclusion and Future Work

Testing concurrent programs is notoriously hard because the behavior of a concurrent program not only depends on input values but also is affected by the way the executions of threads are interleaved. There is often an exponentially large number of interleavings that need to be explored and an exhaustive search is mostly infeasible. The research in this thesis focused on heuristic-based test generation techniques for concurrent programs. We proposed a set of techniques that use heuristics to reduce the exploration space by focusing on a manageable subset of inputs and interleavings.

Summary

The first approach that we took to alleviate the exploration problem was to ignore input exploration and explore the interleaving space under fixed inputs. However, the interleaving space itself is huge enough for real world concurrent programs to make the complete exploration infeasible. Therefore, many techniques that follow this approach (e.g., prediction techniques) target specific types of bugs and try to select and explore a subset of the interleavings that have more chances to reveal those bugs. These techniques have focused mostly on bugs corresponding to atomicity violations, data races, and assertion violations. However, there are other types of bugs (e.g., memory bugs, deadlocks, etc.) that have not been studied in this approach and

are worth to be considered.

In Chapter 2, we introduced a pattern, called null reads, for targeting memory bugs that lead to null-pointer dereferences in concurrent programs; i.e., interleavings that realize this pattern are good exploration candidates to find null-pointer dereferences. We developed a prediction technique that according to a single observed execution of the program, predicts other runs (under fixed inputs) that realize a null read pattern. We studied two different encodings of the prediction problem, one as a set of logical constraints and one as an AI planning problem. The former allows us to use SMT solvers to search for solutions while the latter enables us to benefit from the compact encoding techniques and advanced heuristic-based searching algorithms embedded in AI planners. Our prediction technique is both sound and scalable. We provided a theorem that proves the soundness. The scalability of the prediction technique is supported by our set of experiments. Our experiments also showed that our prediction technique is very fast and effective in finding null-pointer dereferences in concurrent programs. Another valuable property of the prediction technique is that it provides a general framework which can be applied on patterns other than null reads to find other types of bugs.

The next approach that we took to alleviate the exploration problem in testing concurrent programs was to explore both input and interleaving spaces of the programs but based on an approximation model. Most of the techniques that follow this approach, use concurrent trace programs, i.e., program slices built based on program executions, as approximation models for concurrent programs. These techniques fix the approximation model a priori and utilize static symbolic analyses to find assertion violation in the approximation model.

In Chapter 3, we developed a test generation technique based on concurrent trace programs. According to a previous research [25], many runtime bugs (including assertion violations) can be encoded as branches in an active testing framework. We used this result and built a multi-trace analysis with the aim of increasing branch coverage in concurrent programs which inherently targets a broader set of bugs than assertion violations. The multi-trace analysis, for each uncovered branch, tries to find an interloper segment from an execution trace that provides a

value needed to cover the branch and searches for input values and a schedule that would cover the uncovered branch by inserting the interloper segment in another run. Our test generation technique combines the sequential concolic testing with the multi-trace analysis by subjecting each thread to sequential concolic testing first to increase branch coverage in individual thread as much as possible. Then, upon saturation multi-trace analysis targets uncovered branches. Furthermore, the test generation technique does not fix the approximation model; it extends the approximation model by each generated test run. Our experiments showed that the test generation technique is very effective in increasing branch coverage in concurrent programs and catching concurrency bugs.

The last approach in test generation for concurrent programs was to consider the programs in the first place and use heuristics for input/interleaving exploration that allow us to provide some meaningful coverage guarantees for the programs. Most of the techniques that follow this approach use context bounding (which is defined based on the notion of control flow among threads) as the heuristic to limit the interleaving exploration. However, many thread interleavings might be equivalent to each other according to the way threads interfere with each other. Therefore, exploring all such interleavings reduces the efficiency without discovering any new bugs.

In Chapter 4, we introduced a new heuristic, called bounded-interference, for input/interleaving exploration in concurrent programs. This heuristic characterizes a subset of interleaving space by a parameter that bounds the number of interferences among the threads. Therefore, it can be used to provide coverage guarantees modulo interference bound. Another property of this heuristic is that it is defined based on the notion of flow of data between threads (in contrast to the control-based notions such as context bounding). Therefore, it can be incorporated into sequential testing techniques to explore the input and interleaving spaces of the concurrent programs in a unified manner.

In Chapter 5, we evaluated the effectiveness of the bounded-interference heuristic by developing a sequentialization technique based on this heuristic. Given a concurrent program

and an interference bound k, our sequentialization transforms the concurrent program into a sequential program such that the resulting sequential program encodes all behaviours of the concurrent program with maximum k number of interferences among the threads. One advantage of this sequentialization is that traditional sequential testing techniques can be applied on the resulting sequential program without any modification. Inputs of the concurrent program and interference scenarios are both encoded as inputs of the resulting sequential program and therefore underlying sequential testing tools are able to explore both input and interference scenario spaces, side by side. We proved that our sequentialization is sound and could be complete (modulo interference bound) depending on the coverage guarantees that the underlying sequential testing tool provides. Our experiments showed that most of concurrency bugs can be revealed by allowing a few number of interferences among the threads and hence bounded-interference is an effective heuristic in finding concurrency bugs.

After ensuring the effectiveness of the bounded-interference heuristic and to avoid the overhead of sequentialization and the dependency of the coverage guarantees on the underlying sequential testing tools, we adapted a sequential concolic testing technique with the bounded-interference heuristic to generate tests for concurrent programs with coverage guarantees. In Chapter 6, we introduced a new component in concolic testing that explores possible interference scenarios, within the interference bound. Then, for each interference scenario, it generates a test (i.e., input values and a schedule) that realizes the interference scenario (if possible). A nice property of this testing technique is that it provides a general framework where one can employ different strategies in exploring inputs and interference scenarios. We implemented a search strategy that targets branch coverage in concurrent programs; i.e., interference scenarios are explored based on uncovered branches in the program. We proved that this technique is both sound and complete (modulo interference bound). The completeness does not depend on coverage guarantees of sequential concolic testing techniques since the search strategy in concolic testing is adapted according to the bounded-interference heuristic which guarantees completeness by nature. Furthermore, in contrast to the existing concolic testing techniques

for concurrent programs where coverage guarantees can be provided only when the testing algorithm terminates (after the exploration is completed according to the underlying heuristic), our concolic testing technique is able to quantify the partial work done and provide coverage guarantees at each point of time (e.g., at the occasion of a timeout) modulo the explored bound.

Conclusion

We developed several heuristic-based techniques for testing concurrent programs. These techniques attack the problem of test generation from different points of views and each of them has its own right of existence. In fact, this is the testing goal that defines which of these techniques should be used for testing. The testing goal can range from simplicity and time efficiency to targeting specific types of bugs and providing coverage guarantees.

Our prediction technique simplifies the testing problem by exploring the interleaving space under fixed inputs. Furthermore, interleaving exploration is directed by targeting a specific type of bugs. Although we have developed our prediction technique by targeting null-pointer dereferences in concurrent programs, our technique provides a framework which can be used to investigate other types of bugs as well. This technique is very simple and therefore, can be used in earlier stages of the program design where the goal is to catch simple bugs that does not require complicated scenarios for input values as fast as possible. It can also be used when the tester has an idea under which input values the program might encounter some problems. Furthermore, since interleaving exploration is performed based on a specific type of bugs, it is the most efficient technique when the goal is to catch a specific type of bugs.

Our test generation technique based on program approximations is more expensive than the prediction technique since it performs input exploration as well as interleaving exploration. This technique does not target any specific type of bugs; it tries to find bugs by increasing branch coverage in concurrent programs. In contrast to the prediction technique, this technique is able to catch bugs that depend on complicated scenarios for input values. During the

software testing process, we suggest to apply prediction techniques first to catch simple bugs with less cost. Then, our test generation technique based on program approximation can be applied to catch bugs that might be overlooked by the prediction techniques because of ignoring input exploration. However, due to approximations, this technique cannot provide coverage guarantees for concurrent programs.

The bounded-interference concolic testing technique, like the test generation technique based on program approximations, tries to find bugs by increasing branch coverage in concurrent programs. However, it is able to provides branch coverage guarantees for concurrent programs (modulo interference bound). Therefore, in the limit all coverable branches are guaranteed to be covered at the end of the testing process. However, this technique is more expensive than the other techniques and hence is best to be used at the last stages of software development to provide quality assurance for the software.

Now, one might wonder why do we need the previous techniques when the bounded-interference concolic testing technique is able to provide full coverage guarantees. The answer is that for many large programs, the time and computation limitations might not let the interference exploration to reach its limit. Therefore, for such programs, the interference bound cannot go beyond a certain bound. In this case, by using the previous techniques, one still has a chance to find bugs that require larger number of interferences.

Future Work

In the following, we mention some interesting open problems of this theses that are worth to be addressed in future:

Future Work on Prediction: Our prediction technique targets null-pointer dereferences in concurrent programs by exploring the interleavings that realize a null read pattern. We believe that there is still lots of room to investigate new violation patterns that target other types of program bugs. For example, some array out-of-bound errors could be detected by a pattern

(e, f) where e is a read from a shared variable indexing an array and f is a write to the same shared variable that writes a value that is out of the array bound. Any run that realizes this pattern, i.e., forcing e to read the value written by f, would lead to an array out-of-bound error. Other interesting patterns would be the ones that target deadlocks in concurrent programs. For example, some simple deadlock situations could be detected by a pattern $(ac_1, ac_2, ac'_1, ac'_2)$ where ac_1 and ac_2 are lock acquisition events of locks l and l' in one thread, respectively, and ac'_1 and ac'_2 are lock acquisition of locks l' and l in another thread, respectively, and ac_2 and ac'_2 are inside the lock blocks corresponding to ac_1 and ac'_1, respectively; any run in which ac_2 and ac'_2 are co-reachable would block both threads. We leave investing these violation patterns for future work.

To guarantee soundness, we utilized the maximal causal model in our prediction technique that requires each read in the predicted run (except the one involved in the null read pattern) to read the same value as it did in the original run. Note that a prediction problem does not necessarily have a solution in the maximal causal model. We proposed a relaxation technique, which deviates from the maximal causal model gradually, by allowing some reads to read different values than what they read in the original run. The relaxation technique searches for a run, realizing the given null read pattern, with the minimum number of relaxed reads while there is no preference on which reads to relax. Runs that are predicted by the relaxation technique are no more guaranteed to be feasible. We believe that a more detailed analysis based on the program source code to detect the set of reads whose values do not affect affect any branch condition could help in predicting more feasible runs by adapting the relaxation technique such that those reads have priorities over others to get relaxed.

We also showed that the prediction problem can be encoded as an AI automated planning problem to benefit from the compact encoding techniques and advanced heuristic-based searching algorithms embedded in AI planners. Note that the encoding employed for sound prediction did not exploit numerics. A variety of planners, including a variant of FF, METRIC-FF [32], plan with numeric fluents. There has also been significant work on planning with numerics

using a diversity of approaches including SAT-encodings (e.g., [33]), and most recently with encodings using so-called Planning Modulo Theories [27], the latter holding great promise for test generation with numeric reasoning but with the computational advantages of domain-independent heuristic search-based planning techniques. In the future, we plan to investigate the applicability of these techniques in our relaxation technique which involves numeric reasoning.

Future Work on Test Generation Based on Approximation Models: Our test generation based on program approximations targets increasing branch coverage in concurrent programs. We developed a search algorithm that prioritizes yet uncovered branches according to the number of attempts that have been made for covering them; branches with less number of attempts have priorities over the others. Then, for the branch with highest priority, the multi-trace analysis enumerates all possible interloper segments, looking for input values and a schedule that would result in covering the branch by inserting the interloper segment into another run. Although our experiments showed that this search strategy is effective in increasing branch coverage and finding concurrency bugs, we believe that the test generation technique can still be improved by realizing other heuristics in the search algorithm. For example, the depth of the uncovered branches in the control-flow graph of the program could also be used as a heuristic to prioritize the branches; branches with less depth have priorities over others since covering them might lead to discovering a larger part of the program. Furthermore, interloper segments could also be prioritized according to different factors. For example, interloper segments with shorter lengths could have priorities over others since they are less restrictive w.r.t. generating feasible tests. Finally, the search algorithm can be adapted to incremental testing where it can skip covering some of the uncovered branches (e.g., according to a list provided by the tester). Another topic for future work would be to investigate more sophisticated and targeted search strategies in the multi-trace analysis.

Future Work on Bounded-Interference Concolic Testing: Our bounded-interference concolic testing technique provides coverage guarantees for concurrent programs (modulo inter-

ference bound). We developed a search strategy where all interference scenarios of degree k (i.e., consisting of k interferences) for all uncovered branches are explored before exploring interference scenarios of degree $k + 1$ (starting at $k = 0$). This search strategy is not guaranteed to always be the most efficient one. For example, a search strategy that focuses on one branch at a time and explores the interference space (increasing the number of interferences to the bound) accordingly might be a more efficient strategy when some branches have priorities over the others, e.g., they correspond to assertion violations. Another issue is the size of the interference scenario space that could get very large even for small bounds for large programs. Our current search strategy enumerates all possible interference scenarios (modulo interference bound). We believe that the search strategy can borrow ideas from partial order reduction techniques [86, 26] in verification to avoid redundant exploration of interference scenarios that have the same effect. However, one has to be very careful with applying reduction techniques to preserve completeness. We leave investigating other search strategies and possible reduction techniques in interference scenario exploration as a research topic for future.

Future Work Based on Bounded-Interference Heuristic: The bounded-interference heuristic can be used in program analysis areas other than test generation. For example, the context bounding heuristic has been used in model checking to develop context-bounded model checkers [61, 8] that verify properties modulo a context bound. We can consider similar application for the bounded-interference heuristic to verify properties modulo an interference bound. As another example, the bounded-interference heuristic can be used in bug localization (i.e., identifying the roots of the bug) and bug fixing [36, 37, 48]. Concurrency bugs often relate to improper communication or synchronization among threads. According to the bounded-interference heuristic, we can find interference scenarios with the minimum number of interferences that lead to a single bug. These interference scenarios can be analyzed automatically to localize the bug. Applying the bounded-interference heuristic in automatic bug localization is another interesting topic for future work.

In Chapter 4, we mentioned that the context bounding heuristic is not so efficient as many

interleavings might be equivalent to each other according to the interferences that exist among the threads. We think that partial order reduction techniques [86, 26] can be combined with context bounding to avoid this inefficiency. However, the partial order techniques would define equivalent classes on the set of interleavings such that interleavings in the same class realize the same interference scenarios. In this case, the combination of partial order reduction techniques with context bounding would result in exploring all interference scenarios within a bounded number of context switches. Implementing this approach and comparing it with the bounded-interference heuristic is a topic for future work.

Bibliography

[1] Jorge A. Baier and Sheila A. McIlraith. Planning with first-order temporally extended goals using heuristic search. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 788–795, 2006.

[2] David Bainbridge, Ian H. Witten, Stefan Boddie, and John Thompson. Stress-testing general purpose digital library software. In *Proceedings of the 13th European Conference on Research and Advanced Technology for Digital Libraries*, ECDL'09, pages 203–214, 2009.

[3] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'10, pages 167–178, 2010.

[4] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE'08, pages 443–446, 2008.

[5] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS'06, pages 322–335, 2006.

[6] Feng Chen and Grigore Roşu. Parametric and sliced causality. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 240–253, 2007.

[7] Feng Chen, Traian-Florin Serbanuta, and Grigore Roşu. jPredictor: a predictive runtime analysis tool for Java. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 221–230, 2008.

[8] Lucas Cordeiro and Bernd Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE'11, pages 331–340, 2011.

[9] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08, pages 337–340, 2008.

[10] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 81–94, 2006.

[11] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[12] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'11, pages 411–422, 2011.

[13] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. *SIGOPS Operating Systems Review*, 37:237–252, 2003.

[14] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. (Conc)^{2}olic Testing. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'13, pages 37–47, 2013.

[15] Azadeh Farzan and P. Madhusudan. Causal atomicity. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 315–328, 2006.

[16] Azadeh Farzan, P. Madhusudan, Niloofar Razavi, and Francesco Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'12, pages 47–56, 2012.

[17] Azadeh Farzan, P. Madhusudan, and Francesco Sorrentino. Meta-analysis for atomicity violations under nested locking. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV'09, pages 248–262, 2009.

[18] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. *Communictions of the ACM*, 53:93–101, 2010.

[19] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI'03, pages 1–12, 2003.

[20] John Foley and Chris Murphy. Q&A: Bill Gates On Trustworthy Computing. InformationWeek, 2002. `http://www.informationweek.com/qa-bill-gates-on-trustworthy-computing/6502378`.

[21] Pranav Garg and P. Madhusudan. Compositionality entails sequentializability. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'11, pages 26–40, 2011.

[22] Alfonso Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173:619–668, 2009.

[23] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'97, pages 174–186, 1997.

[24] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'05, pages 213–223, 2005.

[25] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Active property checking. In *Proceedings of the 8th ACM & IEEE International Conference on Embedded Software*, EMSOFT'08, pages 207–216, 2008.

[26] Patrice Godefroid and Pierre Wolper. A partial approach to model checking. *Information and Computation*, 110:305–326, 1994.

[27] Peter Gregory, Derek Long, Maria Fox, and J. Christopher Beck. Planning modulo theories: Extending the planning paradigm. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, ICAPS'12, pages 65–73, 2012.

[28] Alex Groce and Willem Visser. Model checking Java programs using structural heuristics. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'02, pages 12–21, 2002.

[29] Patrik Haslum and Alban Grastien. Diagnosis as planning: Two case studies. In *Proceedings of the International Scheduling and Planning Applications Workshop*, SPARK'11, pages 37–44, 2011.

[30] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492, 1990.

[31] Jöerg Hoffmann. FF: The Fast-Forward planning system. *AI Magazine*, 22:57–62, 2001.

[32] Jöerg Hoffmann. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research (JAIR)*, 20:291–341, 2003.

[33] Jöerg Hoffmann, Carla Gomes, Bart Selman, and Henry Kautz. SAT encodings of state-space reachability problems in numeric domains. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, IJCAI'07, pages 1918–1923, 2007.

[34] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Query-driven program testing. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 151–166, 2009.

[35] Jeff Huang and Charles Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the 2011 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'11, pages 144–154, 2011.

[36] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI'11, pages 389–400, 2011.

[37] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 221–236, 2012.

[38] Saurabh Joshi, Shuvendu K. Lahiri, and Akash Lal. Underspecified harnesses and interleaved bugs. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'12, pages 19–30, 2012.

[39] Vineet Kahlon, Franjo Ivanăić, and Aarti Gupta. Reasoning about threads communicating via locks. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, pages 505–518, 2005.

[40] Henry A. Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, IJCAI'99, pages 318–325, 1999.

[41] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[42] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and precise detection of concurrency errors in systems code using SMT solvers. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV'09, pages 509–524, 2009.

[43] Zhifeng Lai, Shing-Chi Cheung, and Wing Kwong Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE'10, pages 235–244, 2010.

[44] Akash Lal and Thomas Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35:73–97, 2009.

[45] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. *SIGPLAN Notices*, 34(8):1–12, 1999.

[46] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. BugBench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[47] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'08, pages 329–339, 2008.

[48] Shan Lu, Soyeon Park, and Yuanyuan Zhou. Detecting concurrency bugs from the perspectives of synchronization intentions. *IEEE Transactions on Parallel and Distributed Systems*, 23(6):1060–1072, 2012.

[49] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting atomicity violations via access-interleaving invariants. *IEEE Micro*, 27:26–35, 2007.

[50] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. *SIGPLAN Notices*, 44(6):134–143, 2009.

[51] Drew V. McDermott. PDDL — The Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control, 1998.

[52] Jesper B. Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. Difference decision diagrams. In *8th Annual Conference of the EACSL on Computer Science Logic*, CSL'99, pages 111–125, 1999.

[53] Madan Musuvathi and Shaz Qadeer. Chess: systematic stress testing of concurrent software. In *Proceedings of the 16th International Conference on Logic-Based Program Synthesis and Transformation*, LOPSTR'06, pages 15–16, 2007.

[54] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Notices*, 42(6):446–455, 2007.

[55] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, 2008.

[56] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., 2004.

[57] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'03, pages 167–178. ACM, 2003.

[58] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'16, pages 135–145, 2008.

[59] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. *SIGPLAN Notices*, 44(3):25–36, 2009.

[60] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS'77, pages 46–57, 1977.

[61] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 93–107, 2005.

[62] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. *SIGPLAN Notices*, 39:14–24, 2004.

[63] Zvonimir Rakamarić. STORM: static unit checking of concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE'10, pages 519–520, 2010.

[64] Niloofar Razavi, Azadeh Farzan, and Andreas Holzer. Bounded-interference sequentialization for testing concurrent programs. In *Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Technologies for Mastering Change*, ISoLA'12, pages 372–387, 2012.

[65] Niloofar Razavi, Azadeh Farzan, and Sheila A. McIlraith. Generating effective tests for concurrent programs via AI automated planning techniques. *Journal of Software Tools for Technology Transfer*, 2013.

[66] Niloofar Razavi, Franjo Ivanăić, Vineet Kahlon, and Aarti Gupta. Concurrent test generation using concolic multi-trace analysis. In *Proceedings of the 10th Asian Symposium on Programming Languages and Systems*, APLAS'12, pages 239–255, 2012.

[67] Jussi Rintanen. Planning with specialized SAT solvers. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence*, AAAI'11, pages 1563–1566, 2011.

[68] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. Generating data race witnesses by an SMT-based analysis. In *Proceedings of the 3rd International Conference on NASA Formal Methods*, NFM'11, pages 313–327, 2011.

[69] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[70] Koushik Sen. Scalable automated methods for dynamic program analysis. In *PhD Dissertation*, 2006.

[71] Koushik Sen. Race directed random testing of concurrent programs. *SIGPLAN Notices*, 43(6):11–21, 2008.

[72] Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Proceedings of the 2nd International Haifa Verification*

Conference on Hardware and Software, Verification and Testing, HVC'06, pages 166–182.

[73] Koushik Sen and Gul Agha. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, 2006.

[74] Traian-Florin Serbanuta, Feng Chen, and Grigore Rosu. Maximal causal models for sequentially consistent systems. In *Proceedings of 3rd International Conference on Runtime Verification*, RV'12, pages 136–150, 2012.

[75] Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. Testing atomicity of composed concurrent operations. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'11, pages 51–64, 2011.

[76] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. Do I use the wrong definition?: DeFuse: definition-use invariants for detecting concurrency and sequential bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'10, pages 160–174, 2010.

[77] Arnab Sinha, Sharad Malik, and Aarti Gupta. Efficient predictive analysis for detecting nondeterminism in multi-threaded programs. In *Proceedings of the 12th Conference on the Theory and Applications of Formal Methods in Hardware and System Verification*, FMCAD'12, pages 6–15, 2012.

[78] Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In *Proceedings of the 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE'11, pages 99–108, 2011.

[79] Nishant Sinha and Chao Wang. Staged concurrent program analysis. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'10, pages 47–56, 2010.

[80] Nishant Sinha and Chao Wang. On interference abstractions. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'11, pages 423–434, 2011.

[81] Fabio Somenzi and Roderick Bloem. Efficient Bchi automata from LTL formulae. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV'00, pages 247–263, 2000.

[82] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'10, pages 37–46, 2010.

[83] Scott D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proceedings of 2nd Workshop on Runtime Verification*, RV'02, pages 143–158, 2002.

[84] Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, 2008.

[85] Salvatore Torre, P. Madhusudan, and Gennaro Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV'09, pages 477–492, 2009.

[86] Antti Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd International Workshop on Computer Aided Verification*, CAV'90, pages 156–165, 1991.

[87] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, pages 238–266, 1995.

[88] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proceedings of the 16th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'01, pages 70–82, 2001.

[89] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *Proceedings of the 2nd World Congress on Formal Methods*, FM'09, pages 256–272, 2009.

[90] Chao Wang, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 328–342, 2010.

[91] Chao Wang, Mahmoud Said, and Aarti Gupta. Coverage guided systematic concurrency testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE'11, pages 221–230, 2011.

[92] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'06, pages 137–146, 2006.

[93] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, 2006.

[94] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. SideTrack: generalizing dynamic atomicity analysis. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD'09, pages 8:1–8:10, 2009.

[95] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. *SIGOPS Operating Systems Review*, 39(5):221–234, 2005.

[96] Wei Zhang, Chong Sun, and Shan Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'10, pages 179–192, 2010.

Appendix A

Examples of the Logical Constraints and AI Planning Encodings in Prediction

Consider the buggy program in Figure 1. `initialize` and `exit` are the methods of the same class with `x` and `y` being the fields of the class of types `myObject` and `int`, respectively. Method `initialize` is for initializing `x` and `y`, which calls a function `func` of `x` after initializing `x` and `y`. Method `exit` reads an input and if the value of the input is greater than zero writes zero and `null` to `y` and `x`, respectively. Now, consider a concurrent program consisting of two threads T_1 and T_2 calling `initialize` and `exit`, respectively.

Suppose that we execute the program with input value 1 and observe a run in which T_1 is executed before T_2. Global trace $p_1p_2p_3p_4p_5p_6p_7p_8q_1q_2q_3q_4$ is the global trace corresponding to this run where $p_1 = (T_1, ac(l))$, $p_2 = (T_1, wt(x, obj))$, $p_3 = (T_1, wt(y, 1))$, $p_4 = (T_1, rd(y, 1))$, $p_5 = (T_1, rel(l))$, $p_6 = (T_1, ac(l))$, $p_7 = (T_1, rd(x, obj))$, $p_8 = (T_1, rel(l))$, $q_1 = (T_2, ac(l))$, $q_2 = (T_2, wt(x, obj))$, $p_3 = (T_2, wt(y, 1))$, and $q_4 = (T_2, rel(l))$. Based on this global trace and using the maximal causal model, we want to predict a run for the *null-WR* pair $\alpha = (q_3, p_7)$.

Here is the set of constraints obtained by the logical constraints encoding proposed in Section 2.4.1:

$\Psi = PO \wedge FC \wedge LC \wedge DC \wedge \alpha\, C$ where

$PO = (t_{init} < t_{p1})(t_{init} < t_{q1})$

$FC = (true)$

```
public void initialize(){
  int a;
  p1: synchronized (this) {
        p2: x = new myObject();
        p3: y = 1;
        p4: a = y;
  p5: }
      if (a > 0){
        p6: synchronized (this){
              p7: x.func();
        p8: }
      }
  }
}
```

```
public void exit(){
  q1: synchronized (this) {
        if (input() > 0) {
          q2: y = 0;
          q3: x = null;
        }
  q4: }
}
```

Figure 1: Code snippet of a buggy program with null-pointer dereference.

$LC = (t_{p_5} < t_{q_1} \vee\ t_{q_4} < t_{p_1})(t_{q_4} < t_{p_6})$

$DC = (t_{q_2} < t_{p_3} \vee\ t_{q_2} > t_{p_4})$

and $\alpha\, C = (t_{p_3} < t_{p_7} \wedge\ t_{p_1} < t_{q_3})$ for realizing α.

This constraint system is satisfiable and $t_{init} < t_{p_1} < t_{p_2} < t_{p_3} < t_{init} < t_{p_4} < t_{p_5} < t_{q_1} < t_{q_2} < t_{q_3} < t_{q_4} < t_{p_6} < t_{p_7}$ defines a schedule which would lead to a null-pointer dereference at event q_3 when the program reads value 1 from the input (as in the observed run).

We also developed a planning encoding for predicting null-pointer dereferences in Section 2.5.2. Here is the set of actions in the planning domain that encode the null-pointer dereference prediction:

(: ACTION Ac_{p_1}

: PRECONDITION (AND (NOT $Done_{p_1}$) ($Available_l$))

: EFFECT (AND ($Done_{p_1}$) (NOT ($Available_l$)))

)

(: ACTION Ac_{p_2}

: PRECONDITION (AND ($Done_{p_1}$) (NOT $Done_{p_2}$))

: EFFECT (AND ($Done_{p_2}$) (x_{p_2}))

)

(: ACTION Ac_{p_3}

: PRECONDITION (AND ($Done_{p_2}$) (NOT $Done_{p_3}$))

: EFFECT (AND ($Done_{p_3}$) (y_{p_3}))

)

(: ACTION $Ac_{p_4}\_coupled_{p_3}$

: PRECONDITION (AND ($Done_{p_3}$) (NOT $Done_{p_4}$) (y_{p_3}))

: EFFECT (AND ($Done_{p_4}$))

)

(: ACTION Ac_{p_5}

: PRECONDITION (AND ($Done_{p_4}$) (NOT $Done_{p_5}$))

: EFFECT (AND ($Done_{p_5}$) ($Available_l$))

)

(: ACTION Ac_{p_6}

: PRECONDITION (AND ($Done_{p_5}$) (NOT $Done_{p_6}$) ($Available_l$))

: EFFECT (AND ($Done_{p_6}$) (NOT ($Available_l$)))

)

(: ACTION Ac_{p_7}

: PRECONDITION (AND ($Done_{p_6}$) (NOT $Done_{p_7}$) (x_{q_3}))

: EFFECT (AND ($Done_{p_7}$) ($Happened_{p_7}$))

)

(: ACTION Ac_{q_1}

: PRECONDITION (AND (NOT $Done_{q_1}$) ($Available_l$))

: EFFECT (AND ($Done_{q_1}$) (NOT ($Available_l$)))

)

(: ACTION Ac_{q_2}

: PRECONDITION (AND ($Done_{q_1}$) (NOT $Done_{q_2}$))

: EFFECT (AND ($Done_{q_2}$) (y_{q_2}))

)

(: ACTION Ac_{q_3}

: PRECONDITION (AND ($Done_{q_2}$) (NOT $Done_{q_3}$))

: EFFECT (AND ($Done_{q_3}$) (x_{q_3}))

)

(: ACTION Ac_{q_4}

: PRECONDITION (AND ($Done_{q_3}$) (NOT $Done_{q_4}$))

: EFFECT (AND ($Done_{q_4}$) ($Available_l$))

)

(: $goal(Happened_{p_7})$)

According to this encoding, sequence Ac_{p_1}, Ac_{p_2}, Ac_{p_3}, Ac_{p_4}, Ac_{p_5}, Ac_{q_1}, Ac_{q_2}, Ac_{q_3}, Ac_{q_4},

Ac_{p_6}, Ac_{p_7} is a solution for the planning problem which implies a bug-triggering schedule for the program.

Appendix B

Proof of Lemma 6.4.2

Proof. The key feature of ISCs generated by Algorithm 3 that proves the lemma, is that they are generated based on feasible program runs. In fact, this lemma does not hold for any arbitrary C. For an interference scenario $S = (V, E, \ell)$, let $G_{T_i}(S)$, $V(S)$, $E(S)$, and $E_I(S)$ represent G_{T_i}, V, E, and E_I in S, respectively. We assume that σ is flattened as a sequence of thread identifiers that shows which thread should be executed at each step. We show that R is feasible and $IS(R)$ is isomorphic to C.

Let $R_k = P(\bar{i}, \sigma[1..k])$ be the partial program run obtained by executing program P according to σ for k steps and let C_k be a sub-interference scenario of C such that for each thread T_i, C_k contains the first m nodes of $G_{T_i}(C)$ if T_i is executed for m steps in $\sigma[1..k]$. C_k contains edges of C for which both involving vertexes are in C_k. Let $\pi_i(C_k)$ be the sequence of labels of the nodes in the path $G_{T_i}(C_k)$ from the root to the leave.

Now, we have to prove that R_k is feasible and $IS(R_k)$ and C_k are isomorphic for all $1 \leq k \leq n$ (where $n = |\sigma|$). To prove the isomorphism of $IS(R_k)$ and C_k, we prove that (1) $G_{T_i}(IS(R_k))$ and $G_{T_i}(C_k)$ are isomorphic for every thread T_i and (2) $E_I(IS(R_k))$ and $E_I(C_k)$ are isomorphic, i.e., there is an interference edge (u, v) in C_k iff $(isom(u), isom(v))$ is an interference edge in $IS(R_k)$, where $isom(n)$ maps a node n in C_k to its isomorphic node in $IS(R_k)$. We prove by the induction on the number of steps of σ.

Induction base: Without loss of generality we assume that thread T_i performs action a in

the first step in R. First of all, R_1 is feasible because the first step cannot be blocked as no blocking synchronization event (like lock acquire) happens before it in R. We know that C is built from feasible executions of P (i.e., $\pi_i(C_1)$ corresponds to a feasible thread-local execution of T_i) and there is no interference in C_1. Furthermore, T_i is deterministic. Therefore, if T_i is reading/writing to a shared variable or acquiring a lock as its first action a in an execution of the program, then it should read/write to the same variable or acquire the same lock, respectively, as its first action in all program executions. If T_i performs a branching action corresponding to a conditional statement as its first action a, then the action of the node in C_1 should also be a branching action corresponding to the same conditional statement. Let $br(\psi)$ be the label of the node in C_1. Without loss of generality we assume that ψ corresponds the branch where the condition of the conditional statement is true. The conditional statement can only depend on inputs (and not any shared variable reads since there is no read from a shared variable before a in T_i). On the other hand, input vector $\bar{i}$ is obtained from $DC(C)$ which forces the condition of the conditional statement to be true. Therefore, action $a = br(\psi)$ in R_1. This proves that $G_{T_i}(IS(R_1))$ and $G_{T_i}(C_1)$ are isomorphic. Also, there is no interference for the first step in both $IS(R_1)$ and C_1 and hence $E_I(IS(R_1))$ and $E_I(C_1)$ are both empty. As a result, $E_I(IS(R_1))$ and $E_I(C_1)$ are isomorphic.

Induction hypothesis: Suppose that for all $1 < k \leq n-1$, R_k is feasible, and $IS(R_k)$ is isomorphic to C_k. Each symbolic variable r corresponding to a read from a shared variable in $IS(R_k)$ is mapped to a symbolic variable r' in C_k, according to the isomorphism.

Induction step: We prove that R_n is feasible, and $IS(R_n)$ is isomorphic to C_n. Without loss of generality, assume that in the n^{th} step, thread T_i performs an action a. We have the following cases according to the type of a:

- $a = wt(x, val)$: Thread T_i is able to perform a since it is not a synchronization action and hence R_n is feasible. We know that $\pi_i(C_n)$ represents a feasible thread-local execution of T_i. According to induction hypothesis, $G_{T_i}(IS(R_{n-1}))$ and $G_{T_i}(C_{n-1})$ are isomorphic meaning that T_i in run R_{n-1} takes exactly the same path as it takes in the

feasible thread-local execution represented by $\pi_i(C_{n-1})$. Together with the determinism of T_i, it is implied that the action corresponding to the last node in $G_{T_i}(C_n)$ is $wt(x, val')$ and val' is equal to val where each symbolic variable r is replaced by its map r'. Therefore, $G_{T_i}(IS(R_n))$ and $G_{T_i}(C_n)$ are isomorphic. Furthermore, a does not introduce any new interference in $IS(R_n)$ since it is the last event in R_n. Therefore, $E_I(IS(R_n))$ = $E_I(IS(R_{n-1}))$ which is, by induction hypothesis, isomorphic to $E_I(C_{n-1})$. On the other hand, $E_I(C_{n-1})$ is equal to $E_I(C_n)$ since all nodes in C_{n-1} are causally before the last node of $G_{T_i}(C_n)$ in C_n and hence the last node in $G_{T_i}(C_n)$ does not involve in any interference in C_n. Therefore, $E_I(IS(R_n))$ is isomorphic to $E_I(C_n)$.

- $a = ac(l)$: Thread T_i is able to perform perform a without getting blocked since R_n follows the schedule σ which is lock consistent according to the temporal-consistency constraints $TC(C)$. Therefore, R_n is feasible. According to induction hypothesis, $G_{T_i}(IS(R_{n-1}))$ and $G_{T_i}(C_{n-1})$ are isomorphic meaning that T_i in run R_{n-1} takes exactly the same path as it takes in the feasible thread-local execution represented by $\pi_i(C_{n-1})$. Together with the determinism of T_i, it is implied that the action corresponding to the last node in $G_{T_i}(C_n)$ is $ac(l)$. Therefore, $G_{T_i}(IS(R_n))$ and $G_{T_i}(C_n)$ are isomorphic. Furthermore, a does not introduce any new interference in $IS(R_n)$. Therefore, $E_I(IS(R_n))$ = $E_I(IS(R_{n-1}))$ which is, by induction hypothesis, isomorphic to $E_I(C_{n-1}) = E_I(C_n)$. As a result, $E_I(IS(R_n))$ is isomorphic to $E_I(C_n)$.

- $a = rel(l)$: Thread T_i is able to perform a since it is not a blocking action and hence R_n is feasible. According to induction hypothesis, $G_{T_i}(IS(R_{n-1}))$ and $G_{T_i}(C_{n-1})$ are isomorphic meaning that T_i in run R_{n-1} takes exactly the same path as it takes in the feasible thread-local execution represented by $\pi_i(C_{n-1})$. Together with the determinism of T_i, it is implied that the action corresponding to the last node in $G_{T_i}(C_n)$ is $rel(l)$. Therefore, $G_{T_i}(IS(R_n))$ and $G_{T_i}(C_n)$ are isomorphic. Furthermore, a does not introduce any new interference in $IS(R_n)$. Therefore, $E_I(IS(R_n))$ = $E_I(IS(R_{n-1}))$ which is, by

induction hypothesis, isomorphic to $E_I(\boldsymbol{C}_{n-1}) = E_I(\boldsymbol{C}_n)$. As a result, $E_I(IS(R_n))$ is isomorphic to $E_I(\boldsymbol{C}_n)$.

- $a = \mathit{br}(\psi)$ corresponding to a conditional statement S: Thread T_i is able to perform a since it is not a synchronization action and hence R_n is feasible. According to induction hypothesis, $G_{T_i}(IS(R_{n-1}))$ and $G_{T_i}(\boldsymbol{C}_{n-1})$ are isomorphic meaning that T_i in run R_{n-1} takes exactly the same path as it takes in the feasible thread-local execution represented by $\pi_i(\boldsymbol{C}_{n-1})$. Together with the determinism of T_i, it is implied that the action corresponding to the last node in $G_{T_i}(\boldsymbol{C}_n)$ should be branching action corresponding to the same conditional statement S in T_i. Let the expression of this branching action be ψ'. Without loss of generality, we assume that ψ represents that the condition of statement S is true. We prove that ψ' represents that the condition of statement S is true as well, i.e., ψ' is equal to ψ where each symbolic variable r is replaced by its map r'. $\bar{i}$ satisfies $\mathit{DC}(\boldsymbol{C})$, and therefore it also satisfies $\mathit{DC}(IS(\boldsymbol{C}_n)) = \mathit{DC}(IS(\boldsymbol{C}_{n-1})) \wedge \psi'$. $\bar{i}$ also satisfies $\mathit{DC}(IS(R_{n-1})) \wedge \psi$ (i.e., conditions on the execution path in R_n). Since $IS(R_{n-1})$ is isomorphic to $IS(\boldsymbol{C}_{n-1})$, therefore any model for $DC(IS(R_{n-1}))$ is a model for $DC(IS(\boldsymbol{C}_{n-1}))$ and vice versa. Therefore, ψ' cannot correspond to the condition of statement S being false. As a result, $G_{T_i}(IS(R_n))$ and $G_{T_i}(\boldsymbol{C}_n)$ are isomorphic. Furthermore, a does not introduce any new interference in $IS(R_n)$. Therefore, $E_I(IS(R_n))$ = $E_I(IS(R_{n-1}))$ which is, by induction hypothesis, isomorphic to $E_I(\boldsymbol{C}_{n-1}) = E_I(\boldsymbol{C}_n)$. As a result, $E_I(IS(R_n))$ is isomorphic to $E_I(\boldsymbol{C}_n)$.

- $a = \mathit{rd}(x, r)$: Thread T_i is able to perform a since it is not a synchronization action and hence R_n is feasible. According to induction hypothesis, $G_{T_i}(IS(R_{n-1}))$ and $G_{T_i}(\boldsymbol{C}_{n-1})$ are isomorphic meaning that T_i in run R_{n-1} takes exactly the same path as it takes in the feasible thread-local execution represented by $\pi_i(\boldsymbol{C}_{n-1})$. Together with the determinism of T_i, it is implied that the action corresponding to the last node in $G_{T_i}(\boldsymbol{C}_n)$ should be equal to $\mathit{rd}(x, r')$ for some symbolic variable r'. Therefore, $G_{T_i}(IS(R_n))$ and $G_{T_i}(\boldsymbol{C}_n)$

are isomorphic. To prove $E_I(IS(R_n)) = E_I(\boldsymbol{C}_n)$ we consider the following cases:

- The leaf of $G_{T_i}(\boldsymbol{C}_n)$, is a node n_r (labeled with (T_i, $\mathit{rd}(x, r')$)) that is involved in an interference edge in $\boldsymbol{C}_n$. Let n_w be a node such that $(n_w, n_r) \in E_I(\boldsymbol{C}_n)$. We prove that there is an interference edge from $isom(n_w)$ to $isom(n_r)$ (i.e., the leaf of $G_{T_i}(IS(R_n))$) in $IS(R_n)$. σ_n is a model for $\mathit{TC}(\boldsymbol{C}_n)$ that orders the nodes in $\boldsymbol{C}_n$. Since, $(n_w, n_r) \in E_I(\boldsymbol{C}_n)$, n_w should be the last node writing to x before n_r, according to σ_n. Since $IS(R_{n-1})$ and $\boldsymbol{C}_{n-1}$ are isomorphic, σ_{n-1} orders the nodes in $IS(R_{n-1})$ such that $isom(n_w)$ be the last node writing to x. Therefore, there is an interference edge from $isom(n_w)$ to $isom(n_r)$ in $E_I(IS(R_n))$. Therefore, $E_I(IS(R_n))$ is isomorphic to $E_I(\boldsymbol{C}_n)$.
- The leaf of $G_{T_i}(\boldsymbol{C}_n)$, is a node n_r (labeled with (T_i, $\mathit{rd}(x, r')$)) that is not involved in any interference edge in $\boldsymbol{C}_n$. Let n_w be the last node before n_w in $G_{T_i}(\boldsymbol{C}_n)$ such that it is labeled with (T_i, $\mathit{rd}(x, val)$) for some val. We prove that $isom(n_r)$ (i.e., the leaf of $G_{T_i}(IS(R_n))$) is not involved in any interference edge in $IS(R_n)$. σ_n is a model for $\mathit{TC}(\boldsymbol{C}_n)$ that orders the nodes in $\boldsymbol{C}_n$. According to σ_n, n_w should be the last node writing to x before n_r. Since $IS(R_{n-1})$ and $\boldsymbol{C}_{n-1}$ are isomorphic, σ_{n-1} orders the nodes in $IS(R_{n-1})$ such that $isom(n_w)$ be the last node writing to x. $isom(n_w)$ is labeled with (T_i, $\mathit{rd}(x, val')$) where val' is equal to val where each symbolic variable r is replaced by its map r'. Therefore, the last write to x is done by thread T_i and hence $isom(n_r)$ is not involved in any interference edge. Therefore, $E_I(IS(R_n)) = E_I(\boldsymbol{C}_n)$.

□

Appendix C

Proof of Lemma 6.4.6

Proof. We prove by induction on k.

Induction Base: When $k = 0$, Algorithm 3 performs traditional sequential concolic testing of individual threads by path exploration. Therefore, for each branch br that is coverable by sequential testing, either it is covered by the initial random test (at line 8), or there exists a corresponding realizable ISC C in W^0 such that either $\mathit{sink}(C) = br$ or $\mathit{sink}(C) = br'$ where br' is in thread $\mathit{Th}(br)$ before br, and the generated test for C covers br. As a result, all writes that can happen without any interference are added to the interference forest after processing W^0.

Induction Hypothesis: For each k-coverable branch br ($1 \leq k \leq n-1$), there exists a realizable ISC C in W^k whose generated test covers br. This implies that each write that require k interferences (where $1 \leq k \leq n-1$) to happen is added to the interference forest while processing W^k.

Induction Step: Let br be an n-coverable branch. We prove that W^n contains a realizable ISC C' whose generated test covers br (C' could have a sink other than br). Let C be a realizable ISC with n interferences and $\mathit{sink}(C) = br$. Suppose that $\alpha = br_{0,1}$, $br_{0,2}$, ..., br_{0,m_0}, $br_{k_1,1}$, $br_{k_1,2}$, ..., $br_{k_1,m_{k_1}}$, $br_{k_2,1}$, ..., $br_{k_h,1}$, ..., $br_{k_h,m_{k_h}}$ is the sequence of branch nodes in $G_{\mathit{Th}(br)}$ where $br_{i,j}$ represents the j^{th} branch node that requires *exactly* i interferences according to C to be covered. Note that the set of interferences required to cover a branch $br_{i,j}$ according

to C might be *a subset of* interferences in the causal interference scenario of $br_{i,j}$ in C; i.e., $i \leq |E_I(CIS(C, br_{i,j}))|$. For example, a read involved in an interference in $G_{Th(br)}$ before $br_{i,j}$ might only affect a branch that comes after $br_{i,j}$ although it appears in $E_I(CIS(C, br_{i,j}))$. According to α, the first m_0 branches are coverable during sequential testing, then the next m_{k_1} branches require (*same*) k_1 interferences, then the next m_{k_2} branches require (*same*) k_2 interferences, and so on, where $0 < k_1 < k_2 < .. < k_h = n$, and $br = br_{k_h,m_{k_h}}$. Let C_{k_1}, C_{k_2}, ..., C_{k_h} be the minimal realizable ISCs (which are subgraphs of C) for branches $br_{k_1,1}$, $br_{k_2,1}$, ..., $br_{k_h,1}$, respectively. Each C_{k_i} is a subgraph of $C_{k_{i+1}}$ for all $1 \leq i < h$.

We first prove (by induction) that W^n contains C_{k_h} at some point during the execution of Algorithm 3. Then, we show that an ISC C' with $sink(C') = br_{k_h,i}$ (for some $1 \leq i \leq m_{k_h}$) which is a super-IS of C_{k_h} with exactly the the same set of interferences in C_{k_h}, is put in W^n at some point during the execution of Algorithm 3, whose generated test covers br. Now, we prove by induction that $W^{k_h} = W^n$ contains C_{k_h} at some point during the execution of Algorithm 3.

Induction Base: We show that W^{k_1} contains C_{k_1} at some point during the execution of Algorithm 3. Since $br_{0,1}$, $br_{0,2}$, ..., br_{0,m_0} are coverable by sequential concolic testing, the algorithm covers them through the initial path exploration. The test that covers br_{0,m_0}, skips $br_{k_1,1}$ since it requires some interferences to be covered. Therefore, a dangling node corresponding to $br_{k_1,1}$ will be added to *forest* and the algorithm generates an ISC I with degree 0 (i.e., $I = CIS(forest, br_{k_1,1})$) and inserts it in W^0. Note that $G_{Th(br_{k_1,1})}(I)$ is equal to $G_{Th(br_{k_1,1})}(C_{k_1})$.

According to C_{k_1}, there should be $k \leq k_1$ reads in $Th(br_{k_1,1})$ in I before $br_{k_1,1}$ that are required to be interfered with writes from other threads. Each of those writes requires $< k_1$ interferences to happen. We order these reads based on the number of interferences their corresponding writes require. Suppose that each read node r_i is interfered with w_i that requires w_i^d interferences (for $1 \leq i \leq k$), and $w_1^d \leq .. \leq w_k^d$. According to the induction hypothesis, each write w_i is added to the interference forest while the algorithm processes $W^{w_i^d}$.

While processing W^0, the algorithm picks and removes I from W^0 at line 12. Since I is not

realizable, it will be added to UN^0. If $w_1^d = 0$ then ExploreISCs (called at line 17) generates an ISC for $br_{k_1,1}$ by using I and introducing an interference from w_1 to r_1. If $w_1^d > 0$ then ExploreISCs (called at line 23) generates the same ISC, while processing $W^{w_1^d}$ and after w_1 has occurred, by selecting I from UN^0 and introducing an interference from w_1 to r_1. In both cases, the generated ISC I' is of degree $w_1^d + 1$ and will be added to $W^{w_1^d+1}$.

Now, either $w_2^d < W^{w_1^d+1}$ or $w_2^d >= W^{w_1^d+1}$. In the first case, the algorithm picks and removes I' from $W^{w_1^d+1}$ (while processing $W^{w_1^d+1}$) at line 12 and generates an ISC I'' by calling ExploreISCs (called at line 17) where an interference is added from w_2 to r_2. In the second case, since I' is not realizable, it will be added to $UN^{w_1^d+1}$ while processing $W^{w_1^d+1}$. While processing $W^{w_2^d}$ and after w_2^d has occurred, the algorithm selects I' from $UN^{w_1^d+1}$ and generates the same ISC I'' by adding an interference is added from w_2 to r_2. In both cases, the resulting I'' will be added to $W^{combined(w_1,w_2)+2}$ where $combined(w_1, w_2)$ represents the number of distinct interferences that are required for both w_1 and w_2. This pattern continues by the algorithm until the last read r_k is interfered and by then the generated ISC is equal to C_{k_1} and will be added to $W^{combined(w_1,..,w_k)+k} = W^{k_1}$.

Induction Hypothesis: W^{k_i} contains C_{k_i} for all $1 < i < h$ at some point during the execution of Algorithm 3.

Induction Step: $W^{k_{h-1}}$ contains $C_{k_{h-1}}$ at some point during the execution of Algorithm 3. While processing $W^{k_{h-1}}$, the algorithm generates a test that realizes $C_{k_{h-1}}$ and leads to covering the branch $br_{k_{h-1},1}$. Assume that this test covers branches $br_{k_{h-1},1}$, ..., $br_{k_{h-1},i}$ for some $1 \leq i \leq m_{k_{h-1}}$.

If $i \neq m_{k_{h-1}}$ then the test skips the branch $br_{k_{h-1},i+1}$. Therefore, a dangling node is added to the *forest* according to $br_{k_{h-1},i+1}$ and an ISC $I = CIS(\textit{forest}, br_{k_{h-1},i+1})$ for $br_{k_{h-1},i+1}$ is generated and added to $W^{k_{h-1}}$ which is also realizable. As a result, while processing $W^{k_{h-1}}$, Algorithm 3 generates a test for ISC I that covers $br_{k_{h-1},i+1}$. Algorithm 3 continues path exploration until all branches $br_{k_{h-1},1}$, ..., $br_{k_{h-1},m_{k_{h-1}}}$ are added to *forest* and covered by some tests.

The test that covers $br_{k_{h-1},m_{k_{h-1}}}$ (could be the test that realized $C_{k_{h-1}}$ in the first place if $i = m_{k_{h-1}}$) would skip $br_{k_h,1}$ since it requires some additional interferences. Therefore, $br_{k_h,1}$ is added as a dangling node in *forest* and the algorithm generates an ISC I with degree k_{h-1} (i.e., $I = CIS(\textit{forest}, br_{k_h,1})$) and inserts it in $W^{k_{h-1}}$. Note that I is a sub-IS of C_{k_h}. According to C_{k_h}, there should be $k \leq k_h = n$ reads in I which are not involved in any interference but are interfered in C_{k_h}. These reads are interfered with writes that require $< n$ interferences to happen. We order these reads based on the number of interferences their corresponding writes require. Suppose that each read node r_i is interfered with w_i that requires w_i^d interferences (for $1 \leq i \leq k$), and $w_1^d \leq .. \leq w_k^d$. According to the induction hypothesis, each write w_i is added to the interference forest while the algorithm processes $W^{w_i^d}$.

While processing $W^{k_{h-1}}$, the algorithm picks and removes I from $W^{k_{h-1}}$ at line 12. Since I is not realizable, it will be added to $UN^{k_{h-1}}$. If $w_1^d \leq k_{h-1}$ then ExploreISCs (called at line 17) generates an ISC for $br_{k_h,1}$ by using I and introducing an interference from w_1 to r_1. If $w_1^d > k_{h-1}$ then ExploreISCs (called at line 23) generates the same ISC while processing $W^{w_1^d}$ after w_1 occurred, by selecting I from $UN^{k_{h-1}}$ and introducing an interference from w_1 to r_1. In both cases, the generated ISC I' is of degree $combined(I, w_1) + 1$ and is added to $W^{combined(I,w_1)+1}$ where $combined(I, w_1)$ represents the number of the distinct interferences that are in I or are required for w_1 to happen.

Now, either $w_2^d < W^{combined(I,w_1)+1}$ or $w_2^d >= W^{combined(I,w_1)+1}$. In the first case, the algorithm picks and removes I' from $W^{combined(I,w_1)+1}$ (while processing $W^{combined(I,w_1)+1}$) at line 12 and generates an ISC I'' by calling ExploreISCs (called at line 17) where an interference is added from w_2 to r_2. In the second case, since I' is not realizable, it will be added to $UN^{combined(I,w_1)+1}$ while processing $W^{combined(I,w_1)+1}$. After w_2^d has occurred while processing $W^{w_2^d}$, the algorithm selects I' from $UN^{combined(I,w_1)+1}$ and generates the same ISC I'' by adding an interference from w_2 to r_2. In both cases, the resulting ISC I'' will be added to $W^{combined(I,w_1,w_2)+2}$ where $combined(I, w_1, w_2)$ represents the number of distinct interferences that are in I or are required for w_1 or w_2. This pattern continues by the algorithm until

the last read r_k is interfered and by then the generated ISC is equal to C_{k_h} and will be added to $W^{k_h} = W^n$.

We have proved that W^n contains C_{k_h} which we know is realizable. Therefore, while processing W^n, the algorithm generates a test for C_{k_h} that covers $br_{k_h,1}$. Assume that this test covers branches $br_{k_h,1}$, ..., $br_{k_h,i}$ for some $1 \leq i \leq m_{k_h}$. If $i = m_{k_h}$ then the proof is complete and $C' = C_{k_h}$ is the ISC in W^n whose generated test covers br. If $i \neq m_{k_h}$ then the test skips the branch $br_{k_h,i+1}$. Therefore, a dangling node is added to the *forest* according to $br_{k_h,i+1}$ and an ISC $I = CIS(\textit{forest}, br_{k_h,i+1})$ for $br_{k_h,i+1}$ is generated and added to W^n which is also realizable. As a result, while processing W^n, Algorithm 3 generates a test for ISC I that covers $br_{k_h,i+1}$. Algorithm 3 continues path exploration until all branches $br_{k_h,1}$, ..., $br_{k_h,m_{k_h}}$ are added to *forest* and covered by some tests. Here as well, W^n contains a realizable ISC C', at some point of time, whose generated test covers $br = br_{k_h,m_{k_h}}$. □