Constraint Solving for Diagnosing Concurrency Bugs

Sepideh Khoshnood

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Chao Wang, Chair

Michael S. Hsiao

Binoy Ravindran

May 1, 2015

Blacksburg, Virginia

Keywords: Concurrency, Bug Localization, Bounded Model Checking, MAX-SAT

Constraint Solving for Diagnosing Concurrency Bugs

Sepideh Khoshnood

(ABSTRACT)

Programmers often have to spend a significant amount of time inspecting the software code and execution traces to identify the root cause of a software bug. For a multithreaded program, debugging is even more challenging due to the subtle interactions between concurrent threads and the often astronomical number of possible interleavings. In this work, we propose a logical constraint-based symbolic analysis method to aid in the diagnosis of *concurrency bugs* and find their root causes, which can be later used to recommend repairs. In our method, the diagnosis process is formulated as a set of constraint solving problems. By leveraging the power of constraint satisfiability (SAT) solvers and a bounded model checker, we perform a semantic analysis of the sequential computation as well as the thread interactions. The analysis is ideally suited for handling software with small to medium code size but complex concurrency control, such as device drivers, synchronization protocols, and concurrent data structures. We have implemented our method in a software tool and demonstrated its effectiveness in diagnosing subtle concurrency bugs in multithreaded C programs.

Dedication

I dedicate this thesis to my mother and my father.

Acknowledgment

I would like to extend my gratitude to my advisor, Dr. Chao Wang, for his consistent guidance and continuous support throughout my research. He has been a tremendous mentor for me and his advice and encouragement have been extremely valuable in helping me complete my thesis. I would also like to thank Dr. Michael Hsiao and Dr. Binoy Ravindran for serving as members of my M.S. thesis committee.

I thank Markus Kusano for being a great colleague and for all his help during evaluation and in collecting test programs. I am grateful for all the help and inspiration I received from my friends in the Reliable Software Systems (RSS) group at Virginia Tech. I also thank all my friends who supported me during these past two years and incented me to strive towards my goal.

Last but not the least, special thanks go to my parents, my brother, and my sister for their endless love, support, and encouragement. They have always been there for me through thick and thin, and have supported me spiritually throughout my life.

Sepideh Khoshnood, May 1, 2015

Contents

List of Figures

List of Tables

List of Abbreviations

BMC	Bounded Model Checking
SAT	Satisfiability
SMT	Satisfiability Modulo Theory
MAX-SAT	Maximum Satisfiability
MUS	Minimally Unsatisfiable Subformula
TF	Trace Formula

Chapter 1

Introduction

Multithreaded programs are notoriously difficult to design and analyze due to the subtle interaction between concurrent threads and the astronomical number of possible interleavings. Because of the inherent complexity, it is often challenging for programmers to reason about the concurrency related behavior of their software code. Testing is also difficult because the program execution is inherently nondeterministic, meaning that reproducing concurrency bugs is hard as a bug may only appear under a specific interleaving of threads. Furthermore, even after a bug is detected, the programmer still needs to sift through the relevant code and inspect the failing execution to localize the root cause. Finally, coming up with a correct repair is a nontrivial task. For example, although a race condition may be eliminated by introducing a critical section or imposing a certain execution order via signal–wait primitives, it is often difficult for programmers to decide which approach is better or if a certain fix itself is bug-free. For all these reasons, having an automated software tool to help identify the potential root cause and suggest possible repairs can be beneficial to programmers.

Over the past decade, a variety of techniques have been proposed for localizing and explaining concurrency bugs. However, some of these existing techniques focused only on bugs involving a single shared variable [19, 62, 49, 59, 58]. Other techniques were able to handle bugs involving multiple shared variables, but they only focused on specific concurrency bug patterns [50, 66]. There are also methods that can detect general concurrency bugs involving both single variable and multiple variables [63, 51]. However, they often do not provide enough diagnosis information to help programmers understand the bug well enough so they are able to fix it effectively afterward.

1.1 Motivation

Our work is inspired by recent developments in logical constraint based methods for diagnosing bugs in sequential software [36, 18, 54]. A representative of these methods is a tool called BugAssist [36], which uses a bounded model checker to systematically search for failing executions, and then a partial maximum satisfiability solver to localize the root cause. The main advantage of this method, as well as similar techniques based on error invariants [18], interpolants [54] and weakest preconditions [72, 71], is the rigorous semantic analysis of the program built upon various constraint solvers. As such, it guarantees that, under realistic assumptions, it can systematically explore all possible failing executions up to a bounded execution depth, thereby providing a comprehensive analysis of the root cause. However, these existing methods only work for sequential programs. It is not immediately clear how the underlying techniques can be extended to handle multithreaded programs.

In this thesis, we fill the gap by extending this technique to concurrent software. We develop a method to effectively localize non-deadlock concurrency bugs and provide programmers with meaningful explanation on what the root cause of the bug is and how they can fix it.

1.2 Contribution

We introduce *ConcBugAssist*, a logical constraint-based symbolic analysis method for diagnosing and repairing concurrency bugs in multithreaded programs. In contrast to the existing methods [36, 18, 54], which focus solely on bugs in sequential programs, our new method focuses solely on concurrency bugs. We assume the sequential program logic is implemented correctly: a sequentialized execution of the program would have the intended behavior. Rather, the concurrency control of the program is buggy: under rare thread schedules, the interleaved execution of the program would exhibit erroneous behaviors. Given such a buggy program, our goal is to identify the root causes of the failing executions automatically.

Figure 1.1 shows the overall flow of our new method. We start with a multithreaded program P where the concurrency bug is a violation of an assertion. First, we apply bounded model checking to compute a failing execution. The failing execution, also called the counterexample, consists of the program input as well as the erroneous thread schedule. The thread schedule imposes a total order over *all* the executed instructions.

Second, we run a partial maximum satisfiability (MAX-SAT) solver to compute a minimum *subset* of the inter-thread ordering constraints that are responsible for the assertion failure. Next, we want

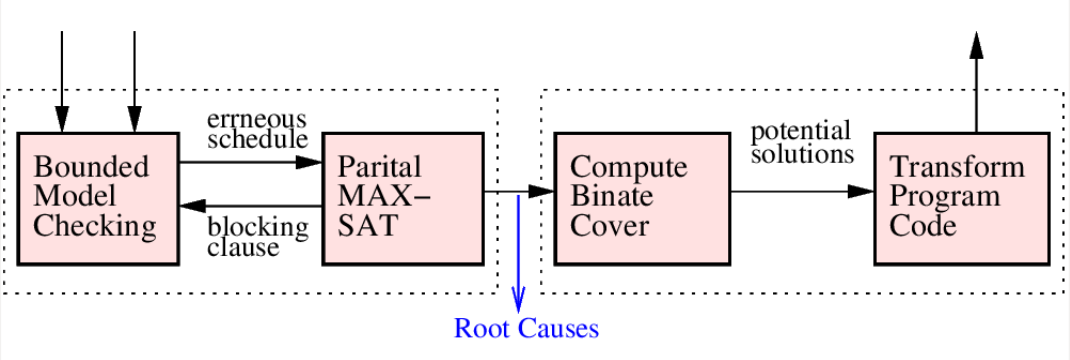


Figure 1.1: ConcBugAssist: The overall *diagnose-and-repair* flow of our new method.

to check for any other thread schedules which result in an error. To do so, we block the previously discovered erroneous thread schedule. To block the schedule, we negate the minimum subset of inter-thread ordering constraints and add them back to the bounded model checker as a *blocking clause*, thus preventing the model checker from selecting this schedule again. After blocking the erroneous schedule, we try to generate a new failing execution. We repeat these steps until no new failing execution can be generated. At this moment, we have computed a set (union) of minimal inter-thread ordering constraints that characterize the root causes of all failing executions.

There are two ways of using the diagnosis result. First, we can help programmers understand the root cause of failure by reporting the diagnosis result. We will show in our experiments that, compared to the full information contained in the failing executions, the set of inter-thread ordering constraints contained in our diagnosis result represent on average a tiny fraction of the ordering constraints in the failing execution. As such, they are much easier to comprehend. Another way to use the diagnosis result is to feed it as input to a follow-up procedure for computing the repairs. By repair we mean modifications to the source code of the original program that are sufficient to

eliminate the assertion violation.

As shown on the right-hand side of Figure 1.1, the full version of *ConcBugAssist* contains a Repair component in which the computation of potential repairs is formulated as an instance of another constraint solving problem (the binate covering problem) based on the result of the Diagnosis component. In this thesis, however, we will not talk about the details of this Repair mechanism as it has not been part of the work done for this thesis. The repair component in *ConcBugAssist* is work of a colleague, therefore, in this thesis, we will only focus on the Diagnosis part of our tool.

It is important to note that, since it is impossible in general, *ConcBugAssist* does not attempt to fully automate the repair process by taking programmers out of the loop. Instead, it aims at leveraging program analysis techniques as a debugging aid to provide meaningful suggestions. There are three reasons to make this choice. First, although we can infer with high certainty the programmer's intent regarding concurrency control, e.g., by analyzing the passing and failing executions using constraint solvers, there is no guarantee that our inference is always correct. In the absence of a complete formal specification, it is generally not possible to automatically repair programs. Second, verifying programs written in realistic programming languages is undecidable in general, and, for concurrent programs, even the context-sensitive synchronization-sensitive analysis of a highly abstracted Boolean program can be undecidable [61]. Third, in practice, developers are generally skeptical about tools that modify software code without going through the standard process of code review and certification.

It is also worth pointing out that the bounded model checker underlying our new method is a light-weight program analysis tool, which means that our method is most suitable for analyzing

the class of multithreaded programs with relatively small code size, but intense thread interactions. Applications in this class include low-level systems code such as device drivers and implementations of concurrent data structures. This is in contrast to methods and tools built upon light-weight static analysis, such as lock-set analysis and control/data flow analysis, which are often more scalable but less accurate than model checking. Tools in the latter class are ideally suited for handling application software with large code size but relatively simple thread interactions.

We have implemented our diagnosis method in a software tool based on the use of the bounded model checker CBMC [40] and a partial MAX-SAT solver called MSUnCore [52]. We have evaluated it on a large set of multithreaded C programs. Our experimental results show that the new method is effective in both localizing the root cause of a concurrency bug and providing meaningful explanations.

To summarize, this thesis makes the following contributions:

- We propose a new symbolic analysis method for diagnosing concurrency bugs by localizing the inter-thread ordering constraints responsible for the manifested failure.
- We provide meaningful explanations about the root cause(s) of the bug to programmers to help them understand the problem to be able to effectively change the program to eliminate erroneous executions.
- We implement the new diagnosis method as a component of the *diagnose-and-repair* framework in a software tool and demonstrate its effectiveness on a set of multithreaded C programs.

1.3 Organization

The remainder of this thesis is organized as follows.

Chapter 2 defines the notations and reviews the basics of model checking concurrent programs. This chapter also reviews some of the existing works on concurrency bug localization.

Chapter 3 introduces the overall flow of our new diagnosis method. This chapter also details the process of diagnosing concurrency bugs in our algorithm.

Chapter 4 provides the implementation details of our diagnosis method. It also describes two optimization techniques we used to improve the runtime performance of our method.

Chapter 5 demonstrates the results of our experimental evaluation and presents two case studies in more details.

Chapter 6 gives our conclusions and outlines the future work.

Chapter 2

Background

In this chapter, we review the basics of bounded model checking (BMC) and maximum satisfiability and then present a model of the concurrent program. We also describe the process of detecting bugs for concurrent programs in modern bounded model checkers.

Specifically, we introduce CBMC, a bounded model checking tool, which focuses on checking safety properties specified using assertion statements throughout an input program. The tool achieves this by modeling the program up to a certain depth as a finite state system and then by looking for assertion violations.

Finally, we go through some of the related work on verifying concurrent programs and existing approaches for localizing root causes of concurrency bugs in multithreaded software.

2.1 Bounded Model Checking

Bounded model checking [8] is a method for checking temporal logic properties in a state transition system by encoding the possible executions as logical formulas and then solving them using constraint solvers. For directly analyzing software code, tools such as CBMC [40] typically focus on checking safety properties specified using assertions. An assertion violation indicates the presence of a bug. To ensure the verification problem remains decidable, bounded model checkers either require the program to be terminating, or ensure the program is terminating by bounding all executions up to a certain depth. Under this assumption, the model checker guarantees that all erroneous executions up to the depth bound are detected. However, if an erroneous execution is beyond the bound, it will be missed by the model checker. Then, to determine if a longer counterexample exists, the input program can be unwound more by increasing the unwinding depth. CBMC also checks that sufficient unwinding is done to make sure that no longer counterexamples can exist by means of unwinding assertions. It is worth noting that the primary goal of bounded model checking is not to verify the correctness of a program but to quickly find bugs.

Since our work uses bounded model checking largely as a black-box, we review only the technical details relevant for understanding our new method. At a high level, bounded model checking relies on a static traversal of the program to encode all possible executions as a set of constraints in logics supported by the underlying solvers. As mentioned earlier, for programs with loops, the conversion from program code to logical constraints involves unrolling the loops up to the bounded depth. The input of the program, to capture all possible values, is represented by symbolic variables. In

the context of multithreaded programs, additional constraints, as defined by the semantics of the program, are constructed to precisely restrict the execution to the set of valid thread schedules.

CBMC reduces the model checking problem to determining the validity of a bit vector equation. It performs a series of transformations on the input program and converts it to a program in static single assignment form (SSA) [14] which only includes branching and assignment statements. In SSA, each variable must be defined and assigned only once. Therefore, if a variable `x` is assigned at multiple locations in the original program, e.g., `x = 5` and `x = 6`, each assignment is associated with a distinct version of `x`. For example, the previous updates to `x` could be transformed to `x1 = 5` and `x2 = 6`. For assignments from different branches of an `if(c)...else...` statement, at the join point, the new version `x3` is defined as `phi(x1,x2)`, representing that `x3` can be, depending on the branch taken, either `x1` or `x2`.

In case of multithreaded programs, within each thread, which is a sequential program by itself, the control and data flow are captured precisely using logical constraints. As described earlier, typically, this involves first converting the program statements to SSA, and then encoding them as logical formulas. Then, to model the concurrency control of the program, CBMC also defines additional logical constraints.

For a comprehensive review of constraint based bounded model checking, refer to Alglave et al. [1] or the CBMC technical report [40].

For the sake of discussing our own work, it suffices to assume that the entire program is statically converted to a logical formula, denoted ϕ, which symbolically captures all valid executions up to a

given depth. To detect violations of a reachability property, e.g., a local assertion, we simply negate the assertion condition p and conjoin it with ϕ. If the combined formula $(\phi \wedge \neg p)$ is satisfiable, then there exists a valid execution of the program where the assertion does not hold. Upon detecting this buggy execution, the solver returns a satisfying assignment mapping each variable in ϕ to a concrete value. Implicitly, the satisfying assignment represents the combination of a concrete program input, a concrete thread schedule, and the sequence of instructions in the failing execution.

The generation of the logical formula for each concurrent program in CBMC will be discussed in more details in Section 2.2.

2.2 Modeling Concurrent Programs

For ease of comprehension, we use the program in Figure 2.1 as an example of bounded model checking for concurrent programs. The program consists of two threads with entry functions `f` and `main`. The `main` thread creates the child thread on Line 8, after which the two threads run concurrently. The two threads share the global variable `x`, whose value is checked in `main` to be non-zero. The `assert` statement on Line 10 indicates that the programmer expects `x` to be a non-zero integer. However, this property may be violated by the program under certain thread schedules.

During bounded model checking, we statically construct the logical formula $\phi \wedge (x == 0)$, where $(x == 0)$ represents the violation of the assertion on Line 10. Furthermore, ϕ, the symbolic representation of the program, can be decomposed into $TF_1 \wedge TF_2 \wedge Ord$, where TF_i, $i \in \{ 1, 2 \}$,

```
0   pthread_t t1;
1   int x = 1;
2
3   void f () {
4     x = 0;
5   }
6
```

```
7   int main () {
8     pthread_create(&t1,0,f,0);
9     if (x != 0)
10      assert(x != 0);
11    return 0;
12  }
13
```

Figure 2.1: Motivating example.

is a *trace formula* representing the sequential execution semantics of the i-th thread. Each instruction in the thread is associated with a *clock* variable representing the logical time when the instruction is executed (i.e., the clock variable imposes a total order over all statements executed by all threads). Finally, to compose the two threads together, we need to restrict the values of the clock variables to ensure only valid thread interactions are allowed (e.g., since a thread cannot execute before it is created, the clock variable of Line 8 must be less than the clock variable of Line 4). These logical constraints are in the Ord formula.

Every satisfying assignment to the above formula corresponds to a possible execution of the program that violates the assertion. In general the satisfying assignment consists of two types of information: (1) a set of concrete values for the program (data) input variables, and (2) a set of concrete values for the *clock* variables, representing the erroneous thread schedule. In the running example in Figure 2.1, since there is no data input, the solver returns only the thread schedule, which is a total order of all instructions visited by the failing execution.

Let $l_1 \rightarrow l_2$ denote that the instruction at Line l_1 is executed before the instruction at Line l_2. For the example in Figure 2.1, one erroneous thread schedule is as follows: $1 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 10$. If the program goes through these instructions in order, `x` would have the value 0 at

Line 10 which violates the assertion.

2.3 Partial Maximum Satisfiability

The logical formulas constructed during bounded model checking are often represented in conjunctive normal form (CNF), where each formula is a conjunction of many clauses, each clause is a disjunction of many literals, and each literal is either a Boolean variable/predicate or its negation. For example, the CNF formula $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3)$ has two clauses $(x_1 \vee \neg x_2)$ and $(x_2 \vee x_3)$, three variables x_1, x_2, x_3, and four literals x_1, $\neg x_2$, x_2, and x_3. In the satisfiability (SAT) problem, we ask whether there exists a satisfying assignment, i.e., a valuation for all variables, such that the entire formula evaluates to true. For the above formula, a satisfying assignment is $\{\, x_1 = \mathsf{true}, x_2 = \mathsf{true}, x_3 = \mathsf{true} \,\}$. If no such valuation exists, we say the formula is unsatisfiable.

The maximum satisfiability (MAX-SAT) problem is a generalization of SAT, with the goal of finding a valuation of all variables that maximizes the number of clauses evaluated to true. If the formula is satisfiable, a solution to the MAX-SAT problem is also a solution to the SAT problem. But, if the formula is unsatisfiable, a solution to the MAX-SAT problem corresponds to the largest subset of clauses that can be satisfied together. The partial MAX-SAT problem is a further extension that separates the clauses into two different categories: *hard* clauses and the *soft* clauses, where the hard clauses must be satisfied and the soft clauses do not have to be satisfied. In the partial MAX-SAT problem, we ask for an assignment that satisfies (1) all hard clauses and (2) as many soft clauses as possible.

There is a duality between the maximally satisfiable subformula returned by a MAX-SAT solver and the *minimally unsatisfiable subformula* (MUS) [43]. The MUS is defined as a subset of the original formula that, by itself, is unsatisfiable, but removing any clause from it would make it satisfiable. In other words, the MUS is an irreducible cause of the infeasibility of the original logical formula. Liffiton et al. [43] show that MUS can be computed by leveraging existing SAT and MAX-SAT solvers [53, 42, 52]. They also show that there may be multiple reasons why a logical formula is unsatisfiable, in which case the removal of any one MUS may not be sufficient to make it satisfiable. When a formula contains multiple MUSs, it will remain infeasible as long as any of the MUSs are present.

There is also another generalization of the satisfiability problem called Weighted Partial MAX-SAT. In this last group, different clauses may have different weights, where the weight to each clause is the penalty to falsify that clause. The idea behind weighted partial MAX-SAT is that not all restrictions are equally important. We should mention that even though we make use of a weighted partial MAX-SAT solver called MiniSat [53], we assign the same weight to all soft clauses as in our work all the soft clauses have the same importance and could be equally responsible for the failure of the counterexample trace.

2.4 Related Work

In the literature there exist multiple works on fault localization. Some of these works perform localization based on multiple runs of a program and compare failing runs with a bunch of successful

runs and after some analysis, they define heuristic metrics on the program traces to detect locations which differentiate failing runs from successful ones. Delta Debugging [77, 13] and Darwin [60] are examples of methods that do such comparisons.

Delta Debugging systematically isolates the relevant parts of input from failing executions by comparing the bad executions with passing executions. By doing this, they narrow down the state difference between good and bad runs and after multiple runs of the program, they reach to the segments of input that can be blamed for the failure of the bad execution. The major difference between our method and these methods is that they require a set of passing traces as an additional input. These passing traces need to be similar to the failing one they are trying to explain and should have large portions in common with the original failing trace. This makes these techniques limited to cases where at least one such passing trace can be found per failing execution.

Another work, DIDUCE [30], computes likely invariants from passing runs and checks for their violations on other runs. Statistical methods [19], on the other hand, compute a suspiciousness factor for statements based on the frequency they occur in successful and failing runs. There are also methods that use symbolic techniques to explain and reason about errors in counterexample traces which were obtained by model checking [6, 29].

Our work was inspired by the set of recent works on using constraint solvers for diagnosing software bugs [36, 18, 54, 60]. However, these methods were designed solely for diagnosing logical bugs in sequential software. None of these methods handle concurrency bugs in multithreaded programs. In contrast, our work focuses primarily on diagnosing concurrency bugs and providing useful explanations. Similar to BugAssist, our method computes a compressed error trace which only

contains information which is related to the presence of bug and can explain it. As mentioned in the previous chapter, this information can further be used to compute potential fixes to eliminate the bug.

Our work is related to methods for synthesizing synchronizations among concurrent threads based on a specification [67] or by making interleaved executions conform to sequential executions [9]. For example, the method proposed by Bloem et al. [9] used a model checker to guide the insertion of atomic regions to force all interleaved executions to behave the same as the sequential execution. They also targeted a certain class of programs where computations in the data flow are largely independent of the concurrency control, where uninterpreted functions could be used to soundly abstract away the data path. In contrast, our method focuses on diagnosing faulty concurrent programs with existing, but potentially buggy, implementations of the concurrency control.

The work by Wang et al. [68, 69] on dynamic deadlock avoidance via discrete control is also related. Their approach relied on building a whole-program Petri-Net model, based on which they applied the theory of discrete control to find ways of healing deadlocks dynamically. However, the method did not handle concurrency bugs other than deadlocks. Liu and Zhang [45] extended the approach to include more bug patterns, e.g., certain types of atomicity violations, but not general concurrency bugs targeted by our new method, which include any non-deadlock concurrency bug that can be modeled as violation of an assertion.

Krena et al. [39, 41, 38] and Jin et al. [35] proposed methods for matching known concurrency bug patterns and fixing them by inserting locks based on predefined rules. They focus on data-races or one-variable, three-access, atomicity violations, but do not handle general concurrency bugs

(e.g., assertion violations), since there are concurrency bugs that cannot be fixed solely by inserting locks. In contrast, our method relies on a more general analysis framework, which has wider application and at the same time requires neither predefined bug patterns nor prescribed repair strategies from the user. There is also a large body of work on diagnosing concurrency failures through dynamic analysis and/or machine learning [11, 76, 32, 59, 74, 7], but cannot systematically detect and diagnose failing executions.

Another difference between our method and the most of the aforementioned static and dynamic analysis techniques is that our method relies on bounded model checking, which is a more precise analysis technique. In general, light-weight static analysis is ideally suited for handling programs with large code size, but infrequent and relatively simple thread interactions, whereas model checking is more suitable for handling programs with a smaller code size, but more complex thread interactions. Examples for the latter case include low-level systems code, device drivers, and implementations of concurrent data structures.

2.5 Summary

This chapter presented a brief overview of bounded model checking and maximum satisfiability as to express the content of the thesis more formally. The idea of modeling concurrent programs by building a logical formula was explained. It was shown that the formula represents the trace formula of each thread and the relationships between events from different threads. The chapter also included the related work on diagnosing concurrency bugs.

Chapter 3

Constraint-Based Method for Diagnosing Concurrency Bugs

This chapter presents our approach for diagnosing concurrency bugs in given erroneous C/C++ programs. First, we describe the process for generating a failing execution. Then, we present the detailed description of our localization algorithm and explain how the result of our diagnosis method can help to eliminate the observed concurrency bug(s). Finally, we go through an example program and apply our approach to find the root cause of the concurrency failure present in it. The overall flow of our method is illustrated in Algorithm 1 which takes the C/C++ source code of a multithreaded program together with its correctness specification as input and returns a minimal set of new constraints whose addition to the original input program would eliminate the discovered bug(s) by making the failing executions infeasible.

As shown earlier in Figure 1.1, our method consists of a diagnosis phase and a repair phase. In the diagnosis phase, given a program P and a property `assert(`p`)`, our goal is to compute the set, ϕ_{Δ}, of minimal inter-thread ordering constraints causing the violation. The set ϕ_{Δ} may be reported directly to the programmers, or it can be used as input to a repair component to compute potential bug fixes.

3.1 Generating the Failing Executions

The first step of the diagnosis method, whose pseudocode is shown in Algorithm 1, leverages the bounded model checker to generate failing executions. The input includes the program P, the assertion condition p, and the maximum execution depth d. The program P can be represented as a deterministic multithreaded program whose behavior is uniquely decided by the pair (in, sch) containing the data input (in) and thread schedule (sch). So, a failing execution is represented by a pair (in, sch) under which the program satisfies the condition $\neg p$ (i.e., the property is violated). Bounded model checkers such as CBMC [40] are ideally suited for systematically generating such failing executions.

Specifically, Algorithm 1 constructs a logical formula, ϕ, to capture all valid executions of the program P up to the given depth d (Line 2). Then, the conjunction $(\phi \wedge \neg p)$ is able to capture all the failing executions symbolically. By examining this formula we will be asking whether there exists any trace to program P in which the the assertion condition p can be violated. If this combined formula is satisfiable (Line 3), then there exists a data input and thread schedule

Algorithm 1 Diagnosing the concurrency failure.

Input: Program P, depth d, and the failed assertion p
Output: Constraint ϕ_Δ to block all failed executions

1: $\phi_\Delta \leftarrow \varnothing$
2: $\phi \leftarrow$ ENCODEVALIDEXECUTIONS(P, d)
3: **while** $(\phi \wedge \neg p)$ is satisfiable **do**
4: $\quad(\phi_{in}, \phi_{sch}) \leftarrow$ GENERATEBADEXECUTION$(\phi \wedge \neg p)$
5: $\quad\phi_{core} \leftarrow$ GENERATEUNSATCORE$(\phi \wedge \phi_{in} \wedge p, \phi_{sch})$
6: $\quad\phi \leftarrow \phi \wedge \neg\phi_{core}$
7: $\quad\phi_\Delta \leftarrow \phi_\Delta \cup \{\phi_{core}\}$
8: **end while**
9: **return** ϕ_Δ

((ϕ_{in}, ϕ_{sch}), Line 4) such that when provided as input to P the condition p is violated. The subroutine GENERATEBADEXECUTION extracts the constraints over the data input and thread schedule from the satisfiable formula $\phi \wedge \neg p$.

At this point, it is worth noting that our focus is on diagnosing concurrency bugs as opposed to logical defects in the sequential computation of the program. That is, the assertion should not be violated under any sequentialized execution, or under every feasible thread schedule. Instead, bugs in the concurrency control logic manifest themselves only under some thread interleavings. If, for example, a program has an assertion violation under all possible thread schedules, it is not concurrency bug but a logical defect in the program, and therefore is out of the scope of this work. To qualify as a concurrency bug, the program must have both passing executions and failing executions under any valid data input (in).

Under this assumption, our goal is to analyze the erroneous thread schedule, ϕ_{sch}, returned by the bounded model checker, and localize the subset of inter-thread ordering constraints that are responsible for the failure. In practice, the number of ordering constraints in ϕ_{sch} may be very large

since it represents a total order of all instructions visited by the failing execution. To make the matter worse, there may be many failing executions as well. Reporting the entire total order, one per failing execution, to the programmers is not only complex, but it is often unnecessary. Doing so may cause confusion and would also require the programmer to spend a longer amount of time to find the cause of the error.

Our focus is to minimize the set of ordering constraints so as to retain only those necessary for explaining the failure.

3.2 Localizing the Ordering Constraints

Next, we continue analyzing the remainder of Algorithm 1. Our procedure for localizing the inter-thread ordering constraints responsible for the failure is shown on Line 5. It takes two sets of constraints: the *hard* constraints ($\phi \wedge p \wedge \phi_{in}$), and the *soft* constraints (ϕ_{sch}), as input and returns a minimal subset (ϕ_{core}) of the ordering constraints in ϕ_{sch} causing the assertion violation as output. We will explain shortly why these constraints are considered as hard and soft.

The subset ϕ_{core} is computed inside the subroutine GENERATEUNSATCORE by first constructing an intentionally unsatisfiable formula, $\phi \wedge p \wedge \phi_{in} \wedge \phi_{sch}$, and then computing its minimal unsatisfiable subformula (MUS).

First, the formula is guaranteed to be unsatisfiable because the conjunction $\phi \wedge p \wedge \phi_{in} \wedge \phi_{sch}$ is a contradiction: the subformula $\phi \wedge \phi_{in} \wedge \phi_{sch}$ restricts the program (ϕ) to the data input and thread

schedule ($\phi_{in} \wedge \phi_{sch}$) which were just determined to cause the program to violate the assertion ($\neg p$ holds). Thus, the conjunction of this formula with p is an unsatisfiable contradiction (it is "asking" the solver if the program can be executed under the buggy input and thread schedule such that the property p holds). Specifically, there is a contradiction because, for a deterministic program, when both the data input and the thread schedule are fixed, the program should either pass or fail the assertion.

Second, the subformula $\phi \wedge \phi_{in} \wedge p$ is guaranteed to be satisfiable because it represents the set of passing executions. Based on the assumption mentioned earlier, there must be at least one passing execution, because otherwise, this is not a concurrency bug since the program would fail under ϕ_{in} regardless of the thread schedule. Therefore, we know that the root cause of the failure resides in the erroneous schedule, ϕ_{sch}, which is a total order of all instructions visited by the failing execution.

Given both subformulas $\phi \wedge p \wedge \phi_{in}$ and ϕ_{sch}, which contradict each other, we would like to compute a minimal subset, ϕ_{core}, of ϕ_{sch} such that the conjunction $(\phi \wedge \phi_{in} \wedge p) \wedge \phi_{core}$ remains unsatisfiable. Therefore, ϕ_{core} is the minimally unsatisfiable subformula (MUS) when $\phi \wedge p \wedge \phi_{in}$ is a hard constraint and ϕ_{sch} is a soft constraint. It represents the *minimal* set of inter-thread ordering constraints that are responsible for the infeasibility and therefore is the root cause of the concurrency failure. (Recall that the MUS, is the minimal subset of the soft constraints such that when conjuncted with the hard constraints the resulting formula is unsatisfiable).

We use the partial MAX-SAT solver MSUnCore [52] to compute ϕ_{core}. MSUnCore accepts two types of logical constraints as input. The first type of constraints are *hard* constraints, which must be satisfied. The second type of constraints are *soft* constraints, which need to be satisfied as many

as possible. In our case, the hard constraint is $(\phi \wedge \phi_{in} \wedge p)$ and the soft constraint is (ϕ_{sch}).

To eliminate the entire equivalence class of erroneous thread schedules represented by ϕ_{core} (i.e., all the thread schedules containing ϕ_{core}), we add the negation of ϕ_{core} back to ϕ on Line 6. This is equivalent to enforcing the constraint $\neg\phi_{core}$ in the original program, which eliminates the previously found failing trace from the set of all valid executions of the program. Because of this, during subsequent iterations, the model checker will never generate a failing execution containing ϕ_{core}. Furthermore, due to the finite number of bounded program executions, Algorithm 1 is guaranteed to terminate. Finally, during any iteration, the set ϕ_{Δ} contains the diagnosis information of all erroneous thread schedules, one (non-negated) ϕ_{core} per schedule, seen so far. In the end, ϕ_{Δ} contains the diagnosis information across all buggy schedules.

3.3 Diagnosing the Running Example

Consider the running example in Figure 2.1, where the first failing execution returned by the model checker corresponds to the line numbers: $1 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 10$. As previously stated, the order in which statements are executed is represented by a clock variable assigned to each instruction. For ease of presentation, let us assume that e_i, where $i \in \{\ 1, 2, \ldots\ \}$, is the *clock* variable associated with the instruction at line i. Let $e_i \rightarrow e_j$ denote that the instruction at Line i happens-before the instruction at Line j (i.e., the clock variable for line i is smaller than the clock variable for line j). Under these assumptions, the failing execution can be represented by ϕ_{sch},

which is a total order of all the visited instructions:

$$\phi_{sch} \equiv (e_1 \rightarrow e_7) \wedge (e_7 \rightarrow e_8) \wedge \ldots$$

However, many of these ordering constraints are not relevant to the root cause of the error. To localize the root cause, we construct an intentionally unsatisfiable formula as follows:

$$\underbrace{TF_1 \wedge TF_2 \wedge Ord}_{\textit{valid executions } (\phi)} \wedge \underbrace{\phi_{in} \wedge \phi_{sch}}_{\textit{failing trace}} \wedge \underbrace{(x \neq 0)}_{\textit{assertion}}$$

The formula is unsatisfiable because $\phi \wedge \phi_{in} \wedge \phi_{sch}$ represents the failing execution, and yet $(x \neq 0)$ requires the assertion condition to hold (a contradiction). Since the program does not have any data input, $\phi_{in} \equiv \mathsf{true}$. By declaring ϕ_{sch} as *soft* constraints and the rest as *hard* constraints, we are able to localize the subset ϕ_{core} of constraints responsible for the failure.

For the running example in Figure 2.1, we get:

$$\phi_{core} \equiv (e_9 \rightarrow e_4) \wedge \neg(e_{10} \rightarrow e_4)$$

When viewed graphically, the root cause clearly shows the lack of atomicity between lines 9 and line 10:

After adding $\neg\phi_{core}$ back to ϕ, we are able to block all the other erroneous executions. In other words, ϕ_{core} implicitly captures an equivalence class of erroneous schedules, all of which share the same core constraints in ϕ_{core}. Although this particular example requires only one iteration in

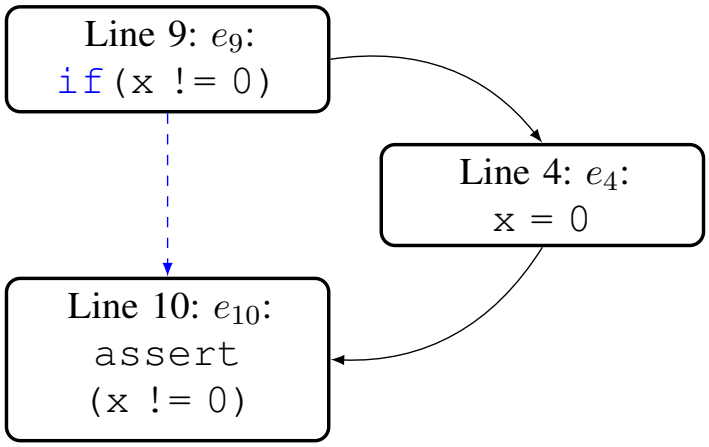


Figure 3.1: Root cause computed for the bug in the motivating example

Algorithm 1, in general, our diagnosis procedure needs multiple iterations to eliminate all erroneous executions, as usually not all erroneous schedules for a program share the same core constraints. Within each iteration, we conjoin $\neg\phi_{core}$ with ϕ. At the same time, we record ϕ_{core} in ϕ_{Δ} for latter use. When the model checker can no longer find failing executions, ϕ_{Δ} contains the set of constraints sufficient for explaining all failing executions.

3.4 Diagnosing the MySQL Example

In this section, we will go through another example. Consider the program shown in Figure 3.2. The two threads, `t1` and `t2`, share variables `x` and `y`. The assertion condition `(x == y)` indicates that the intended behavior is for the assignment statements in both threads to run atomically, without interference from the other thread. However, this atomicity property is not enforced properly in either thread: `t1` can interleave in between `t2`'s updates and vice versa. As a result, there are two

```
1  int x = 0;
2  int y = 0;
3
4  void f1(void) {
5    x = 0;
6    y = 0;
7  }
8
9  void f2(void) {
10   x = 1;
11   y = 1;
12 }
13
```

```
14
15
16 int main() {
17   pthread_t t1, t2;
18   thread_create(t1,f1);
19   thread_create(t2,f2);
20
21   thread_join(t1);
22   thread_join(t2;
23   assert(x == y);
24   return 0;
25 }
26
```

Figure 3.2: Buggy program with atomicity violations between the two threads.

equivalence classes of erroneous schedules: one where `x = 0` is immediately followed by `y = 1` and another where `x = 1` is immediately followed by `y = 0`.

Algorithm 1, presented earlier in this section, would be able to return the localized constraints for both equivalence classes (ϕ_{core_1} and ϕ_{core_2}) of all erroneous schedules:

1. ϕ_{core_1}: $e_{10} \rightarrow e_5 \wedge e_6 \rightarrow e_{11}$,
2. ϕ_{core_2}: $e_5 \rightarrow e_{10} \wedge e_{11} \rightarrow e_6$.

Here, when a constraint such as $e_i \rightarrow e_j$ appears in ϕ_{core}, it means the happens-before edge is necessary for explaining why the assertion is violated.

To provide a better understanding of the relevance of the two UNSAT cores and the buggy schedules, we construct a graphical representation of each erroneous schedule, consisting of not only the constraints in the UNSAT core, but also the related program-order constraints. Each program-order constraint, denoted $e_i \rightarrow e_i'$, represents the sequential execution order of instructions from the same thread. Figure 3.3 shows the graphical representations of these two erroneous thread schedules side by side. Each interleaving results in an assertion violation. Specifically, Figure 3.3a shows `t1`

writing to `x` (e_5) followed by `t2` writing to `x` (e_{10}). Next, `t2` writes to `y` (e_{11}) before `t1` (e_6). This results in a final state where `x == 1` and `y == 0`. Figure 3.3b shows a similar schedule where the final state results in `x == 0` and `y == 1`.

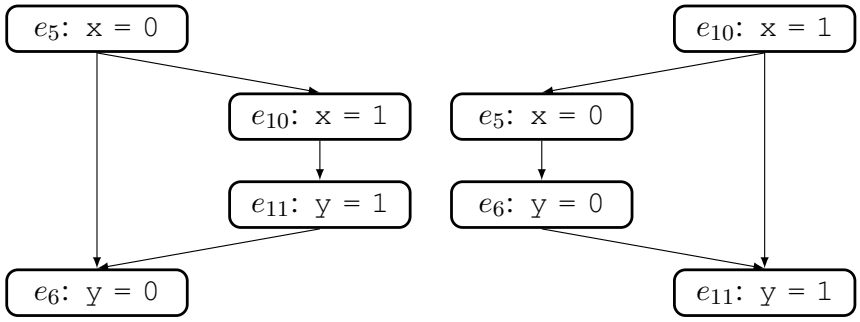


Figure 3.3: Graphical representation of the two erroneous interleavings of Figure 3.2.

Next, we pass the UNSAT cores computed by our diagnosis method to a repair component. This repair component computes a set of new happens-before constraints such that enforcing any of them in the original program is sufficient to prevent the erroneous interleaving. In the repair method, a new happens-before relation is defined: ($\rightarrow_s$) where $e_i \rightarrow_s e_j$ indicates that in all schedules of the program e_i occurs before e_j. Given an erroneous schedule, such as in Figure 3.4a, the repair method adds new happens-before edges to create cycles in the graph.

Intuitively, inserting such a cycle creates a contradiction ensuring that the interleaving cannot occur. For example, in the erroneous interleaving in Figure 3.4b, there is an edge $e_5 \rightarrow e_{10}$. Thus, a cycle can be created by inserting a new happens-before edge $e_{10} \rightarrow_s e_5$. This creates a proof by contradiction ensuring that the interleaving does not happen. The reason is that in order for the erroneous interleaving to occur, e_5 must occur before e_{10}, but, at the same time, e_{10} must always

occur before e_5, leading to a contradiction.

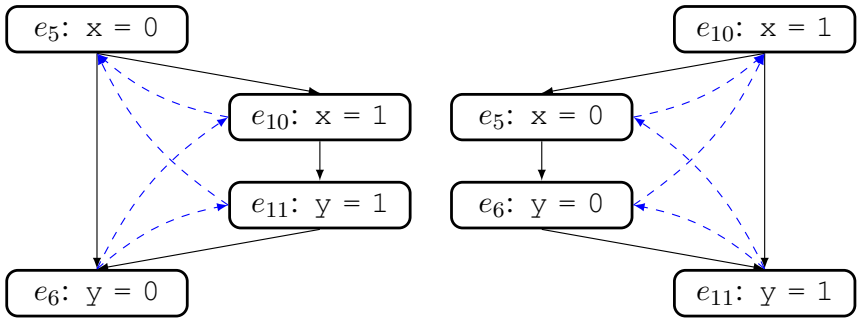


Figure 3.4: Potential happens-before edges to block the erroneous interleavings in Figure 3.2.

Figure 3.4 shows all the possible new happens-before edges (dashed edges) that, individually, can block the erroneous schedule. The solid edges, in contrast, are the ordering of the erroneous schedule. The solid edges are the ordering of the erroneous interleaving while the dashed edges are the happens-before edges which block the erroneous interleaving. It is interesting to note that some of the dashed edges are negations of the solid edges, such as $e_6 \rightarrow_s e_{11}$ and $e_{10} \rightarrow_s e_5$. However, there are also dashed edges, such as $e_6 \rightarrow_s e_{10}$ and $e_{11} \rightarrow_s e_5$, that cannot be constructed directly from the negations of the solid edges: they can only be constructed using the graph based algorithm in the repair component of ConcBugAssist. For a comprehensive explanation of the repair method, please refer to [37].

3.5 Summary

In this chapter, we proposed an approach based on model checking and partial maximum satisfiability to diagnose the root cause of concurrency bugs present in a given C/C++ program. In the next

chapter, we will proceed to some of the implementation details of our diagnosis method and will present two optimization techniques with some preliminary results on a set of benchmark programs.

Chapter 4

Implementation and Optimizations

In this chapter, we present the implementation details of our method. We also illustrate two optimization techniques we used for improving the runtime performance of our diagnosis algorithm. Finally, we present some preliminary results for these optimization techniques and make a comparison.

4.1 Implementation based on MSUnCore

As mentioned earlier in Chapter 3, by default, we use MSUnCore, a partial MAX-SAT solver, to compute the UNSAT cores for the counterexample traces.

Recall that our diagnosis method is implemented as an extension to the latest version of the CBMC [40] model checker, which supports the verification of multithreaded programs [1]. By default, CBMC uses MiniSat, a state-of-the-art Boolean SAT solver, to determine validity of a

given input program's formula. The logical formula CBMC generates for the input program in this case is a Boolean formula in which every data variable, condition, assumption, and assertion is represented as a vector of bits. This means that this equation captures the behavior of the program at a bit-precise level. As a result, every literal in this formula is a 1-bit variable and the clauses are disjunction of a few of these literals.

Consider the MySQL example in Figure 3.2. By default, CBMC generates a Boolean formula with 5456 clauses and 1683 literals for that example. As mentioned before, this generated formula models all the possible executions of that program together with the input variables and the assertions. Every assignment to the variables of this formula corresponds to a new valid execution of the program. Using MiniSat again, CBMC asks for an assignment set which violates the assertion in that program. Having such an assignment, which captures a specific execution, our goal is to look for a passing execution which is very similar to that counterexample trace and is only different from it in a small portion of its cross-thread scheduling.

For our ease, we added a new type of constraint to the CBMC-generated formula for modeling the cross-thread scheduling of the input program. We call these new constraints happens-before relationships. These constraints are added between any two statements, if they are in different threads, both accessing a shared variable, and at least one of them is a *Write*. We have two types of happens-before relationships: *hbs* and *hbh*. *hbs* constraints are those in which both statements are coming from the user-written code. *hbh* constraints, on the other hand, are those in which at least one of the two statements comes from built-in functions and/or predefined libraries. For the MySQL example, our tool adds a total of 18 happens-before constraints to the program formula.

When looking for the root cause of the assertion violation in the found counterexample, since we want to enforce its scheduling (so that we could end up with a very similar passing execution), we need to encourage the MSUnCore partial MAX-SAT solver to go through the same thread scheduling, but at the same time make only minor changes in the thread scheduling to make the assertion condition pass this time. That is why we mark *hbs* constraints as soft clauses in the decision procedure. This gives the solver the freedom to play around with the thread scheduling and change it when it is necessary. To tell MSUnCore which clauses are soft and which are hard, we assign a weight equal to 1 to soft clauses and a very large weight to hard ones. The solver looks at these weights as the amount of punishment for not satisfying the clause.

We should mention that MUS by definition is a minimum subset of clauses, from an unsatisfiable formula, whose conjunction is still unsatisfiable and removing any of the clauses from that subset would make it satisfiable. The MUS typically would contain both soft and hard constraints. However, what partial MAX-SAT solvers like MSUnCore actually compute does not exactly match this definition of MUS. Instead, MSUnCore returns a subset of the soft constraints whose value's change would make the whole formula satisfiable. So, in order to compute the MUS, we had to design an algorithm. We call that algorithm *ComputeUnSATCore*.

As you recall, what we were looking for was a minimum set of soft clauses who were conflicting with each other or the hard constraints. So, every time we get a set of clauses from MSUnCore, we temporarily mark those as hard clauses and invoke MSUnCore again on this new formula. This time, a new set of clauses will be returned which are now conflicting with the newly introduced hard clauses. Repeating this process a few times would take us to a state where the whole formula

becomes unsatisfiable. This means that we have already found the whole MUS and the reason why the formula has become UNSAT is we have enforced the whole MUS by marking all of its clauses as hard constraints. At that point, since we have kept track of the clauses we marked as hard, we would know what the MUS is.

Recall that there were two counterexample traces to MySQL example. At the beginning, CBMC finds the counterexample shown in Figure 3.3a. When we fed the extended formula to MSUnCore for the first time, it returns the following clause as the MUS:

$$2049 \wedge \neg 2050$$

which contains two *hbs* constraints 2049 and 2050 meaning:

2049: (e1 = [tid1] line 5 function $f1$ Write $c :: x$) $<$ (e2 = [tid2] line 10 function $f2$ Write $c :: x$)

2050: (e1 = [tid1] line 6 function $f1$ Write $c :: y$) $<$ (e2 = [tid2] line 11 function $f2$ Write $c :: y$)

To block this ordering between lines 5, 6, 10, and 11, we compute the negation of the MUS and add it back to the program formula as a *hard* clause. Remember that we do not want the same ordering to happen in any valid execution. Adding this new constraint makes a contradiction between this new hard constraint and the previous MUS, which was a soft constraint. Therefore, the solver, from now on, is forced to evaluate the new constraint to true and the previous MUS to false.

By adding a new constraint, we blocked at least one failing execution. However, there might exist other counterexample traces which do not fail due to the same root cause. So, we repeat the process again. Checking the updated version of the program formula to the solver returns

the counterexample shown in Figure 3.3b this time. And feeding this new counterexample to MSUnCore returns the following MUS:

$$\neg 2049 \wedge 2050$$

Like before, to block this counterexample, we add its negation back to the program formula. Doing so, results in the new version of the program not having any valid failing executions. At this point, our algorithm terminates and reports the two MUS clauses as the root causes for the assertion violations. This means these new clauses are sufficient for explaining the observed bugs.

Unfortunately, the Boolean formula generated for MiniSat, even from a small C program, can be huge. That would cause the satisfiability check by Minisat and later by MSUnCore to take considerable amount of time compared to the total runtime of the algorithm. In fact, in our method, once we generate an unsatisfiable formula, finding an MUS by MSUnCore is the bottleneck.

The reason is that although the formula given to MiniSat was quite big as well and the extended unsatisfiable formula is also about the same size, finding the MUS would be much more time-consuming as now the solver needs to explore many possible subsets of the clauses in the unsatisfiable formula to find a solution that maximizes the number of satisfied soft clauses. To put it simply, it is not a problem of deciding whether the formula is satisfiable or not anymore. Rather, it is the problem of finding an optimal solution.

This performance issue calls for a need to optimizing our method. In the next section, we will describe the work we have done so far in that regard.

4.2 Optimization Techniques

We propose two optimizations for improving the runtime performance of our diagnosis method. In the remainder of this section, we present these two optimization techniques in details.

4.2.1 Using SMT Solvers

CBMC can be customized to work with different SAT or SMT (Satisfiability Modulo Theory) solvers such as MiniSat [53], Z3 [16], and Yices [17]. Depending on the type of the solver chosen, CBMC generates a logical representation of the input program either in form of a Boolean formula or a formula in a more expressive language beyond conjunction and disjunction of literals, with use of SAT solvers or SMT solvers, respectively. Using a SAT solver would result in more precision, however, using an SMT solver would make it easier to understand the generated logical formula but at the same time, due to the higher level of abstraction, it would result in better performance.

As mentioned in the previous section, in the original implementation, we have been using SAT solvers to make decisions on satisfiability of a given logical formula and to compute UNSAT cores for the unsatisfiable ones. Then, in order to improve our algorithm, we decided to try migrating to SMT solvers from SAT solvers.

The idea behind Z3, and generally all SMT solvers, is the trade-off between precision and performance. SMT solvers are extensions of SAT solvers that check the satisfiability of formulas built from Boolean variables and operations. SMT is the problem of determining whether a first-order formula is satisfiable with respect to some decidable first-order theory. An SMT instance

is a formula in first-order logic, where some functions and predicate symbols have additional interpretations. It could be defined as a SAT instance only with the difference that some of the binary variables are replaced by predicates over a suitable set of non-binary variables. A predicate is basically a binary-valued function of non-binary variables such as equality, uninterpreted functions, arithmetic, etc.

Since Z3 performs the satisfiability check at a higher level of abstraction, it requires a smaller logical formula as input when compared to Boolean SAT solvers. It also causes an increase in the runtime performance of searching for a solution. Therefore, to get the potential boost in performance, we switched the usage of SAT solvers in our algorithm to Z3 SMT solver.

Once again, consider the MySQL example in Figure 3.2. To be able to use Z3 SMT solver, CBMC generates a first-order logical formula in smt2 format with 496 constraints and 240 variables for this example. Using Z3, CBMC looks for an assignment set for all the variables in the formula which violates the assertion in the program. This assignment set gives us a counterexample trace. In a Z3 formula if a constraint is followed by the ”named” keyword, it will be traceable in the computed MUS. So, to create the extended unsatisfiable formula, we attach to the original program formula some *assert* constraints for the program assertion conditions and the values of all input variables and happens-before variables in the counterexample trace and only name those happens-before variables which we want to mark as *soft* constraints. This way, if any of these clauses appears in the MUS, we would be able to know.

The *ComputeUnSATCore* algorithm is a little bit different for Z3 compared to the one we explained for MSUnCore. Z3 does not guarantee to compute a ”minimum” unsatisfiable core. Instead, it

computes an unsatisfiable core which contains the MUS inside of it, but might contain several other constraints as well. So, in this case, we would need to shrink the UNSAT core computed by Z3. How we do it is whenever we get an UNSAT core from Z3, we check the clauses inside the UNSAT core to see whether they actually are an essential part of the MUS. So, each time we remove only one of those clauses from the original formula and we check the satisfiability of the formula. If it is still UNSAT, even without that one clause, it means that clause is not part of the MUS. But if it becomes SAT, it means that clause should be inside the final MUS, as removing it also removed the UNSAT core from the formula. After we check all the clauses from the UNSAT core that Z3 generated, we would know which ones should necessarily be in the MUS.

Going back to the MySQL example, after having generated the extended formula, we invoke Z3 to compute an MUS. Since the first counterexample was the one shown in Figure 3.3a, Z3 gives us the following constraint as the MUS:

(assert (or (not $|memory\_model :: choice\_hbs79|$) $|memory\_model :: choice\_hbs78|$))

which contains two *hbs* variables $|memory\_model :: choice\_hbs79|$ and $|memory\_model :: choice\_hbs78|$ meaning:

$$|memory\_model :: choice\_hbs78|:$$

(e1 = [tid1] line 5 function $f1$ Write $c :: x$) $<$ (e2 = [tid2] line 10 function $f2$ Write $c :: x$)

, and

$$|memory\_model :: choice\_hbs79|:$$

(e1 = [tid1] line 6 function $f1$ Write $c :: y$) $<$ (e2 = [tid2] line 11 function $f2$ Write $c :: y$)

Adding the negation of this MUS to the program formula and feeding this updated formula to Z3 will now give us the counterexample shown in Figure 3.3b. After we build a new unsatisfiable formula based on this second counterexample trace and invoke Z3 on it, we will get a new MUS:

(assert (or $|memory\_model :: choice\_hbs79|$ (not $|memory\_model :: choice\_hbs78|$)))

Since there were no other failing executions to the MySQL program except for these two, after we update the formula once again by adding the negation of the second MUS to it, Z3 would not be able to find any other executions which violated the assertion. Then, our diagnosis algorithm terminates successfully having computed a set of those ordering constraints sufficient for explaining the bugs.

Although using Z3 gave us much smaller and actually human-readable logical formulas, and showed improvement in performance on most of the benchmark programs, our experiments showed that in some cases our tool was not able to compute a root cause for the present bug using Z3. The symptom we observed was that after a few iterations of *ComputeUnSATCore* algorithm, the UNSAT core computation by Z3 failed to find an UNSAT core for the unsatisfiable formula. For instance, this is what happened to *Lazy01* program. In the first iteration of our diagnosis algorithm, ...

So far, we believe that due to loss of precision (caused by abstraction), in some cases the formula generated by our method using Z3 did not contain all of the constraints needed to find the root causes of the bugs. This means on those cases, the formula generated had abstracted away some

of the details of the behavioral logic of the input program, some of those that were necessary for explaining the observed bug.

Having encountered this issue, we know that the Z3 version of our implementation still requires extra tailoring and needs to be rethought again to become able to generate an unsatisfiable formula that contains all the necessary constraints for modeling every valid input program, while still keeping it an abstraction. Therefore, this is a part of our work which is still in progress.

4.2.2 Using Boolean Abstraction

In general, scalability is a very important issue with modern concurrent software model checkers. Other than the *data space explosion* that is common even in sequential software, concurrent programs in particular suffer from state explosion due to the exponential growth of number of valid thread executions. *Predicate abstraction* [28] is a well-known technique for dealing with the state explosion problem in software verification [5]. In predicate abstraction, an overapproximation of the program state is computed from the values of a finite number of predicates over the program variables. The method turns a C program into a finite state Boolean program.

For those reasons, we expected that applying the diagnosis method to a Boolean abstraction of the program instead of the C program itself, could increase scalability of our algorithm. So, we decided to use SATABS, a well-known abstraction tool, as a preprocessor to transform our C/C++ input programs into Boolean programs. Our experiments showed that this can speed up the process of diagnosis a little bit. However, our ultimate goal is to apply this technique on a collection of Linux

device drivers which are usually huge and more complex. Right now, we are able to apply our tool directly to Boolean programs for smaller cases, which itself is an advantage anyway. And, we are continuing the work on generating Boolean abstractions of Linux device drivers.

For this optimization technique there are also some failure cases like the benchmark program *VectPrime02* [9] where our tool is not able to compute a root cause for the bug. The symptom here is the same as what we saw earlier for Z3 version of our tool, where after a few iterations, the UNSAT core computation fails and MSUnCore is not able to find an MUS for the unsatisfiable formula.

4.2.3 Experiments

Table 4.1 shows the summary of our experimental results for applying the described optimization techniques on some of the benchmark programs. The characteristics of each benchmark program can be found in Table 5.1. Column 1 shows the name of the benchmark program. Columns 2,3, and 4 indicate the runtime of the diagnosis algorithm under the MSUnCore implementation, Z3 implementation, and applying the MSUnCore implementation on a Boolean version of the program, respectively. As we can see, using the Z3 SMT solver and the Boolean Abstraction technique decreases the runtime of our algorithm. However, on some of the programs, the optimization techniques resulted in the algorithm not being able to compute the root cause of the bug eventually. The symptom observed in these cases has been explained in the sections 4.2.1 and 4.2.2. A ✗ under *Z3* or *Boolean Abstraction* columns shows the occurrence of this symptom.

As our optimizations are still work in progress and based on the result of our experiments on

Table 4.1: Preliminary performance optimization results.

Name	MSUnCore	Z3	Boolean Abstraction
testc	0.7	0.7	0.4
read_write	121.0	14.8	85.3
barrier	5.5	✗	4.6
lazy01	11.9	✗	10.7
VectPrime02	2.4	1.9	✗
mysql-12848	2.5	✗	2.0
mysql-3596	1.1	0.8	0.8
mysql-644	1.0	0.8	0.7
llvm-8441	17.4	✗	16.8
transmission-1.42	1.2	✗	0.9
mozilla-61369	0.8	0.6	0.5

optimization techniques, in the next chapter we will present the experimental results only for the original MSUnCore implementation of our approach, which is a more stable version of the diagnosis method.

Chapter 5

Evaluation

In this chapter, we first show our experimental results on a set of benchmark programs and then present two case studies to illustrate the use of our tool in diagnosis and repair of concurrency bugs. Finally, we discuss some limitations of our tool.

We have evaluated our method on 34 benchmark programs. Our experimental evaluation was designed to answer the following research questions:

- Can our diagnosis method accurately localize the root cause of a concurrency bug?
- Can the repair component compute meaningful code modifications to eliminate the bug, given inputs which are computed by our diagnosis algorithm?

To answer these questions, in the rest of this chapter we first describe the benchmark programs used in the evaluation and then present the detailed results.

Table 5.1: Characteristics of the benchmark programs.

Name	LOC	Threads	Bug Type	Origin
boop	98	3	atomicity violation	[64]
testc	19	2	order violation	[64]
fibbench	47	3	order violation	[64]
fibbench_longer	45	3	order violation	[64]
reorder	105	5	order violation	[64]
account	58	4	order violation	[64]
read_write	140	5	order violation	[64]
barrier	85	4	order violation	[64]
lazy01	55	4	data race	[64]
VectPrime02	183	3	data race	[9]
lineEq2t01	58	3	data race	[9]
linux-tg3	115	3	order violation	[9]
linux-iio	87	3	atomicity violation	[9]
mysql-169	27	3	atomicity violation	[75, 57]
mysql-12848	142	2	atomicity violation	[75, 56]
mysql-3596	83	3	order violation	[48]
mysql-644	165	3	order violation	[48]
apache-21287	79	3	atomicity violation	[2]
apache-25520	192	3	data race	[3]
freebsd-aa	104	4	order violation	[73]
cherokee-0.9.2	188	3	atomicity violation	[75]
llvm-8441	244	3	order violation	[47]
gcc-25330	86	3	atomicity violation	[23]
gcc-3584	104	3	data race	[24]
gcc-21334	94	3	data race	[22]
gcc-40518	114	3	data race	[25]
transmission-1.42	78	3	order violation	[75]
glib-512624	98	3	atomicity violation	[27]
jetty-1187	74	3	order violation	[34]
mozilla-61369	68	3	order violation	[48]
hash_table	156	3	atomicity violation	[31]
list_seq	122	3	atomicity violation	[31]
counter_seq	41	3	data race	[31]
queue_seq	97	3	data race	[31]

5.1 Benchmarks

Table 5.1 shows the statistics of the 34 benchmark programs, including the name, the number of lines of code, the number of threads, and the type of the concurrency bug. The last column also shows the origin of the program. Our benchmarks can be classified into four groups.

The first group consists of the POSIX threads related buggy programs from the 2015 Software Verification Competition [64] (SVCOMP). Although these programs are small in terms of the lines of code, they implement tricky concurrency protocols and synchronization algorithms such as read–write locks.

The second group consists of four buggy programs used by Bloem et al. [9], where the first two are synthetic benchmarks, while *linux-iio* and *linux-tg3* are real bugs found recently in the industrial I/O subsystem (IIO) of the Linux kernel[1], and Broadcom Tigon3 (TG3) Ethernet driver[2], respectively.

The third group consists of bug patterns extracted from various versions of open source applications. They are reported in MySQL [55], the Apache Web Server [4], the FreeBSD Operating System [20], the Cherokee Web Server [10], the LLVM Compiler Framework [46], the GNU Compiler Collection [21], the Linux Kernel [44], the Transmission BitTorrent client [65], the GNOME Library [26], the Jetty HTTP Server [33], and Mozilla's XPCOM library [70]. These programs are used to evaluate the effectiveness of our method in handling the diverse set of bugs from the real world.

The fourth group consists of implementations of concurrent data structures as described in the

[1] http://git.io/JjCEXg
[2] http://git.io/7wWrKw

Art of Multiprocessor Programming book [31]. Some of these programs are stripped off the synchronization operations intentionally to see if our tool can correctly repair them back to normal.

5.2 Experimental Results

Table 5.2 summarizes the results. Columns 1 and 2 show the program name and the diagnosis time, respectively. The experiments were run on a machine with a 2.60 GHz Intel Core i5-3230M CPU and 8 GB of RAM running a 64-bit Linux OS.

Column 3 shows the number of iterations required to complete the diagnosis, i.e., the number of equivalence classes of erroneous schedules. It is also the same as the number of blocking constraints (ϕ_{core}) computed by our method as part of the diagnosis result. Column 4 shows, on average, the number of inter-thread ordering constraints present in an erroneous schedule (ϕ_{sch}); they are the number of constraints that programmers have to inspect manually if they do not use our diagnosis method.

Columns 5–6 show the average size of the root cause returned by our method, in terms of the number of inter-thread ordering constraints to block an erroneous schedule (ϕ_{core}), as well as the the total number of such unique constraints for blocking all erroneous schedules. Finally, Column 7 shows the reduction ratio, i.e., the number of constraints in the root cause divided by the average number of constraints in a bad schedule.

Overall, our method can quickly identify the root cause: most of the programs took only a few

seconds to complete, with the maximum run time of just over two minutes. Furthermore, the reduction ratio in Column 7 indicates that our method is effective in localizing the root cause of a concurrency failure. On average, the number of inter-thread ordering constraints reported in the root cause is significantly smaller than the total number of raw constraints in the error traces returned by CBMC.

The reason why the number of unique constraints for *glib-512624*, *jetty-1187*, *list-seq*, and *queue-seq* appears to be lower than expected is because some happens-before edges are mapped to the same lines of code for their source and target nodes. In such cases, we merge these happens-before edges into one for ease of comprehension.

We also confirmed manually that all the diagnosis results computed by our tool correctly could explain the bugs in the benchmark programs. Furthermore, the root causes were always straightforward to understand. In addition, we will show later in this section that the diagnosis results are specific enough that they can be leveraged to automatically compute a repair using a repair mechanism.

5.3 Case Studies

Finally, we present two case studies to illustrate the use of our tool in diagnosing and repairing concurrency bugs.

***list_seq*:** This is a sequential array based list implementation (i.e., it has no enforced synchronization)

Table 5.2: Summary of the error diagnosis results.

Name	Time (s)	Iter.	Size of Root Cause			Red. Ratio
			Constr./ (ϕ_{sch})	Constr./ (ϕ_{core})	Unique Constr.	
boop	1.2	1	34	2.0	2	5.9%
testc	0.7	1	4	2.0	2	50.0%
fibbench	36.0	2	93	7.5	15	16.1%
fibbench_longer	106.6	2	123	9.0	18	14.6%
reorder	19.9	15	30	4.0	9	30.0%
account	6.4	3	95	1.3	3	3.2%
read_write	121.0	28	76	8.2	27	35.5%
barrier	5.5	9	48	1.6	6	12.5%
lazy01	11.9	2	186	2.0	4	2.2%
VectPrime02	2.39	2	31	3.0	3	9.7%
lineEq2t01	4.83	2	30	4.0	7	23.3%
linux-tg3	5.6	1	98	2.0	2	2.0%
linux-iio	2.5	5	31	4.4	8	25.8%
mysql-169	1.1	2	10	2.0	2	20.0%
mysql-12848	2.5	4	10	4.0	4	40.0%
mysql-3596	1.1	1	13	1.0	1	7.7%
mysql-644	1.0	1	7	2.0	2	28.6%
apache-21287	1.7	2	23	1.5	3	13.0%
apache-25520	7.9	16	23	4.0	4	17.4%
freebsd-aa	22.4	49	27	3.0	12	44.4%
cherokee-0.9.2	6.9	11	34	3.1	4	11.8%
llvm-8441	17.4	21	46	3.3	10	21.7%
gcc-25330	1.2	2	21	1.0	2	9.5%
gcc-3584	1.8	4	19	3.0	3	15.8%
gcc-21334	6.4	1	244	2.0	2	0.8%
gcc-40518	1.4	2	27	2.0	4	14.8%
transmission-1.42	1.2	2	8	1.5	2	25.0%
glib-512624	8.7	17	32	1.3	4	12.5%
jetty-1187	1.8	2	33	1.0	1	3.0%
mozilla-61369	0.8	1	5	1.0	1	20.0%
hash_table	112.0	44	94	1.4	4	4.3%
list_seq	13.6	18	60	1.2	4	6.7%
counter_seq	1.2	2	13	3.0	3	23.1%
queue_seq	7.6	2	135	1.0	1	0.7%
Average	16.01	8.15	51.85	2.77	5.26	16.81%

```
1  struct list {
2  int arr[MAX_SIZE];
3  size_t open;
4  } gl;
5
6  void list_add(list_t *s, int i) {
7  s->arr[s->open] = i;
8  s->open += 1;
9  }
10 void t1_main() {
11 int val;
12 val = 1;
13 list_add(&gl, val);
14 return;
15 }
16 void t2_main() {
17 int val;
18 val = 2;
19 list_add(&gl, val);
20 return;
21 }
22 int main() {
23 thread_t t1, t2;
24 thread_create(&t1, t1_main);
25 thread_create(&t2, t2_main);
26 thread_join(t1);
27 thread_join(t2);
28 assert(list_contains(&gl, 1)
29 && list_contains(&gl, 2));
30 return 0;
31 }
32
```

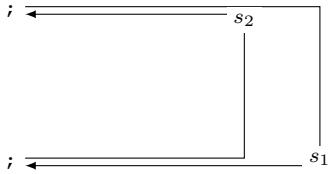


Figure 5.1: Buggy *list_seq* with two potential repairs (s_1 and s_2)

used concurrently by multiple threads. A shortened version of its source code can be seen in Figure 5.1. Thread 1 inserts the item `1` into the list while thread 2 inserts the item `2`. The `main` thread checks that the list contains both `1` and `2` after both threads finish.

The bug is the lack of atomicity in the `list_add()` function: the insertion of an item (Line 7) is not atomic with the update of the lists size (Line 8). Our diagnosis procedure returns this as an explanation: the bug occurs if thread 1 executes Line 7 followed by thread 2 executing Line 8, and thread one executing Line 7 after thread 2 executes Line 8 (and vice versa). In this case, since the value of `open` has not been updated, thread 2 (resp. 1) overwrites the value inserted into the list by

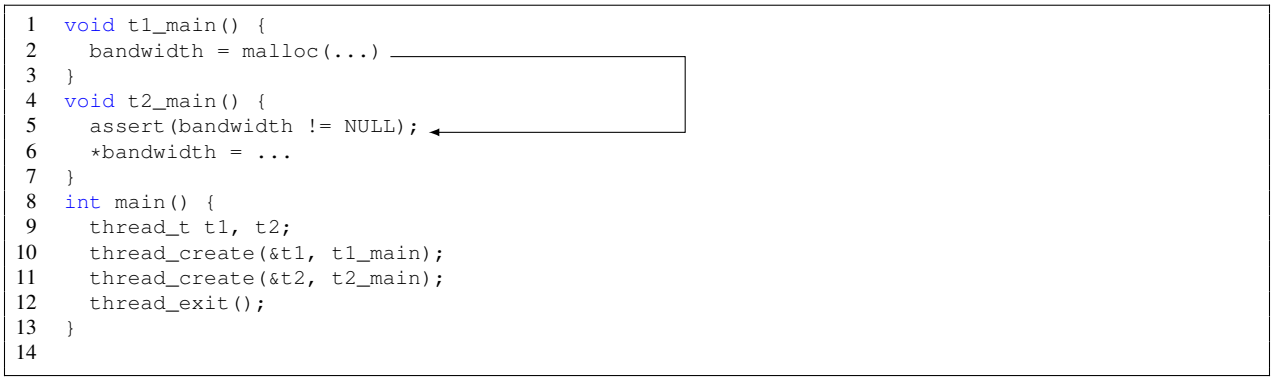
thread 1 (resp. 2). The end result is a list without the value `1` (resp. `2`) so the assertion on Lines 28–29 will fail.

Figure 5.1 also shows two of the potential repairs: s_1 and s_2. The edges are a happens-before constraint which, when added to the program, will prevent the bug from happening. Repair s_1 states that thread one should add first followed by thread two; Repair s_2 is the reverse fix. Together, these two solutions create this solution: either thread 1 can go first or thread 2 can go first. Interestingly, the final result is that our *diagnosis-n-repair* procedure automatically synthesized a concurrent list from a sequential one.

Transmission-1.42: This is a BitTorrent client that contained a data race. The relevant portion of the source code can be seen in Figure 5.2. Two threads share a variable `bandwidth`. The bug is caused by an incorrect assumption that, when thread 2 accesses `bandwidth`, thread 1 will have already initialized it. The root cause of the bug identified by the diagnosis algorithm is Line 5 executing before Line 2.

This means the bug can be prevented by enforcing that Line 2 is always executed before Line 5. Therefore, the repair component will report the repair shown in Figure 5.2 to the user.

We have manually confirmed that this is a correct repair. In fact, this is the actual solution used by developers in the Transmission source to fix the bug.



```
1  void t1_main() {
2    bandwidth = malloc(...)
3  }
4  void t2_main() {
5    assert(bandwidth != NULL);
6    *bandwidth = ...
7  }
8  int main() {
9    thread_t t1, t2;
10   thread_create(&t1, t1_main);
11   thread_create(&t2, t2_main);
12   thread_exit();
13 }
14
```

Figure 5.2: Relevant portion of the code in *transmission-1.42*.

5.4 Discussions

There could be two threats to usefulness of the repairs computed by our tool and its potential for being generalized as a commercial software. The first threat is whether the repairs computed by our method are always correct or not. Another threat is the degree to which the benchmark programs we used in our experiments represent realistic applications. We will discuss each of these threats briefly in this section.

Since we rely on CBMC to find bugs and to validate potential repairs, in general, we cannot guarantee that the generated repairs are always correct repairs. They are merely suggestions to aid programmers in fixing the bugs. In terms of deadlocks, our full tool (diagnosis + repair) will not introduce certain types of deadlocks, i.e., deadlocks caused by the incompatibility of our newly added happens-before constraint(s) and the original thread program order constraints. However, it is still possible for a repair to introduce other type of deadlocks, e.g., from reversed lock ordering between threads. This is due to the fact that CBMC is incomplete for the purpose of detecting

deadlocks. For a comprehensive description of how the repair component of ConcBugAssist works, please refer to [37].

A possible remedy to this issue is to follow up our tool's potential-repair computation with a static deadlock analysis procedure, to weed out the obviously problematic repairs, before presenting them to the programmer.

Again, because our method relies on CBMC as the underlying verification procedure, this can limit its ability of handling large programs. However, this scalability problem may be addressed in two ways. First, the Boolean SAT solvers used in the diagnosis phase may be replaced by SMT solvers [15], which tend to work on higher levels of abstractions and therefore are potentially more scalable than Boolean SAT solvers. Second, our diagnosis method may be applied to a Boolean abstraction of the program, created using well-known predicate abstraction tools such as SATABS [12], as opposed to the concrete program. This may result in some precision loss due to the use of predicate abstraction, but at the same time will significantly boost the runtime performance. As mentioned earlier in Chapter 4, we have been working on applying these optimizations, however, the work is still in progress. We leave further exploration of such optimizations for future work.

Even though scalability is an issue with the work right now, since the BMC procedure is heavy weight, we see potential uses of our tool to be in small but complex concurrent software.

Chapter 6

Conclusions

In this thesis, we have presented a constraint based method for diagnosing concurrency bugs in multithreaded programs by localizing a small set of happens-before constraints sufficient for explaining the root causes. We have also shown that our diagnosis method's output can be used later to compute potential program repairs by iteratively adding additional happens-before constraints to block the erroneous thread schedules. Our new method has been implemented in a software tool, ConcBugAssist, and has been evaluated on a set of multithreaded C programs. Our experiments show that the proposed method is effective in diagnosing and explaining the root cause of concurrency bugs.

The main objective of this thesis work is to develop an automated software tool that can formally verify a multithreaded C/C++ program and detect the root causes of present concurrency bugs. Based on our experimental results, the causes computed by our diagnosis technique can effectively

explain the bugs and when put together with a repair component, can suggest correct and practical fixes. The main bottleneck of our approach is state space explosion, meaning the state space increases exponentially for higher number of threads. Since we use CBMC as the underlying verification procedure, our method's scalability is limited by the CBMC tool. Also, the presented method is limited by the fact that modern model checkers can only verify bounded executions of programs to make sure that the number of valid executions is finite. Therefore, it can only detect concurrency bugs up to the specified bound and will miss those bugs which only happen in longer execution traces.

We believe that the automated concurrency bug diagnosis algorithm can be further optimized to improve scalability. The scalability problem in our work can be addressed in two ways. First, by replacing the Boolean SAT solvers used in diagnosis method with SMT solvers (which tend to be more scalable). Second, by applying the diagnosis method to a Boolean abstraction of the problem, as opposed to the concrete program itself. We have been working on these optimizations and have got some preliminary results, however, the work is still in progress. The further exploration of these performance optimizations will be left for future work.

Bibliography

[1] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *International Conference on Computer Aided Verification*, pages 141–157, 2013.

[2] Apache bug 21287 URL: http://issues.apache.org/bugzilla/show_bug.cgi?id=21287.

[3] Apache bug 25520 URL: https://issues.apache.org/bugzilla/show_bug.cgi?id=25520.

[4] Apache http server project URL: http://httpd.apache.org/.

[5] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI'01)*, pages 203–213, June 2001.

[6] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 97–105, 2003.

[7] M. T. Befrouei, C. Wang, and G. Weissenbacher. Abstraction and mining of traces to explain concurrency bugs. In *International Conference on Runtime Verification*, pages 162–177, 2014.

[8] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207. Springer, 1999. LNCS 1579.

[9] R. Bloem, G. Hofferek, B. Könighofer, R. Könighofer, S. Ausserlechner, and R. Spork. Synthesis of synchronization using uninterpreted functions. In *International Conference on Formal Methods in Computer-Aided Design*, pages 35–42, 2014.

[10] The cherokee web server URL: http://cherokee-project.com/.

[11] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *International Symposium on Software Testing and Analysis*, pages 210–220, 2002.

[12] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In *In TACAS, volume 3440 of LNCS*, pages 570–574. Springer, 2005.

[13] H. Cleve and A. Zeller. Locating causes of program failures. In *ACM/IEEE International Conference on Software Engineering*, 2005.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[15] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340, 2008.

[17] B. Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, 2014.

[18] E. Ermis, M. Schäf, and T. Wies. Error invariants. In *International Symposium on Formal Methods*, volume 7436, pages 187–201. 2012.

[19] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[20] The freebsd project URL: http://www.freebsd.org.

[21] The gnu compiler collection URL: https://gcc.gnu.org/.

[22] Gcc bug 21334 URL: http://gcc.gnu.org/bugzilla/show_bug.cgi?id=21334.

[23] Gcc bug 25530 URL: http://gcc.gnu.org/bugzilla/show_bug.cgi?id=25330.

[24] Gcc bug 3584 URL: http://gcc.gnu.org/bugzilla/show_bug.cgi?id=3584.

[25] Gcc bug 40518 URL: http://gcc.gnu.org/bugzilla/show_bug.cgi?id=40518.

[26] The glib reference manual URL: https://bugzilla.gnome.org/show_bug.cgi?id=512624.

[27] glib bug 512624 URL: https://bugzilla.gnome.org/show_bug.cgi?id=512624.

[28] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV'97)*, pages 72–83. Springer, 1997. LNCS 1254.

[29] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 2005.

[30] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, pages 291–301, 2002.

[31] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[32] N. Jalbert and K. Sen. A trace simplification technique for effective debugging of concurrent programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 57–66, New York, NY, USA, 2010. ACM.

[33] Jetty servlet engine and http server URL: http://www.eclipse.org/jetty/.

[34] Jetty bug 1187 URL: http://jira.codehaus.org/browse/JETTY-1187.

[35] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 389–400, 2011.

[36] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 437–446, 2011.

[37] S. Khoshnood, M. Kusano, and C. Wang. Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs. In *International Symposium on Software Testing and Analysis*, 2015.

[38] B. Krena, Z. Letko, Y. Nir-Buchbinder, R. Tzoref-Brill, S. Ur, and T. Vojnar. A concurrency testing tool and its plug-ins for dynamic analysis and runtime healing. In *International Conference on Runtime Verification*, pages 101–114, 2009.

[39] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pages 54–64. ACM, 2007.

[40] D. Kroening and M. Tautschnig. CBMC—C bounded model checker. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391, 2014.

[41] Z. Letko, T. Vojnar, and B. Krena. AtomRace: data race and atomicity violation detector and healer. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, page 7. ACM, 2008.

[42] C.-M. Li, Z. Fang, and K. Xu. Combining MaxSAT reasoning and incremental upper bound for the maximum clique problem. In *IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 939–946, Nov 2013.

[43] M. H. Liffiton, Z. S. Andraus, and K. A. Sakallah. From Max-SAT to Min-UNSAT: Insights and applications. Technical Report CSE-TR-506-05, University of Michigan, 2005.

[44] The linux kernel archives URL: http://kernel.org.

[45] P. Liu and C. Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *International Conference on Software Engineering*, pages 299–309, 2012.

[46] The llvm compiler infrastructure URL: http://llvm.org/.

[47] Llvm bug 8441 URL: http://llvm.org/bugs/show_bug.cgi?id=8441.

[48] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008.

[49] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[50] B. Lucia, L. Ceze, and K. Strauss. Colorsafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[51] B. Lucia, B. P. Wood, and L. Ceze. Isolating and understanding concurrency errors using reconstructed execution fragments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[52] J. Marques-Silva. The MSUnCore MAXSAT Solver.

[53] Minisat URL: http://minisat.se/.

[54] V. Murali, N. Sinha, E. Torlak, and S. Chandra. What gives? A hybrid algorithm for error trace explanation. In *International Conference on Verified Software: Theories, Tools and Experiments*, pages 270–286, 2014.

[55] Mysql the world's most popular open source database URL: http://www.mysql.com/.

[56] Mysql bug 12848 URL: http://bugs.mysql.com/bug.php?id=12848.

[57] Mysql bug 169 URL: http://bugs.mysql.com/bug.php?id=169.

[58] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Architectural Support for Programming Languages and Operating Systems*, pages 25–36, 2009.

[59] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. In *International Conference on Software Engineering*, pages 245–254, 2010.

[60] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. DARWIN: an approach to debugging evolving programs. *ACM Trans. Softw. Eng. Methodol.*, 21(3):19, 2012.

[61] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[62] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[63] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do i use the wrong definition?: DefUse: definition-use invariants for detecting concurrency and sequential bugs.

In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 160–174, 2010.

[64] SV-COMP. 2015 software verification competition. URL: http://sv-comp.sosy-lab.org/2015/, 2015.

[65] Tranmission URL: https://www.transmissionbt.com/.

[66] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 334–345, 2006.

[67] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 327–338, 2010.

[68] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 281–294, 2008.

[69] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. A. Mahlke. The theory of deadlock avoidance via discrete control. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 252–263, 2009.

[70] Mozilla xpcom URL: https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM.

[71] Q. Yi, Z. Yang, J. Liu, C. Zhao, and C. Wang. Explaining software failures by cascade fault localization. *ACM Transactions on Design Automation of Electronic Systems*, 2015.

[72] Q. Yi, Z. Yang, J. Liu, C. Zhao, and C. Wang. A synergistic analysis method for explaining failed regression tests. In *International Conference on Software Engineering*, 2015.

[73] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 26–36, New York, NY, USA, 2011. ACM.

[74] E. Yom-Tov, R. Tzoref, S. Ur, and S. Hoory. Automatic debugging of concurrent programs through active sampling of low dimensional random projections. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 307–316, 2008.

[75] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *International Symposium on Computer Architecture*, pages 325–336, 2009.

[76] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. *SIGPLAN Not.*, 45:143–154, March 2010.

[77] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. ACM.