ConcBugAssist: Constraint Solving for Diagnosis and Repair of Concurrency Bugs

Sepideh Khoshnood
ECE Dept., Virginia Tech
Blacksburg, VA, USA
sepidehk@vt.edu

Markus Kusano
ECE Dept., Virginia Tech
Blacksburg, VA, USA
mukusano@vt.edu

Chao Wang
ECE Dept., Virginia Tech
Blacksburg, VA, USA
chaowang@vt.edu

ABSTRACT

Programmers often have to spend a significant amount of time inspecting the software code and execution traces to identify the cause of a bug. For a multithreaded program, debugging is even more challenging due to the subtle interactions between threads and the often astronomical number of interleavings. In this work, we propose a logical constraint based symbolic analysis method to aid in the diagnosis of concurrency bugs and to recommend repairs. Both diagnosis and repair are formulated as constraint solving problems. Our method, by leveraging the power of satisfiability (SAT) solvers and a bounded model checker, performs a semantic analysis of the sequential computation as well as thread interactions. The constraint based analysis is designed for handling critical software with small to medium code size, but complex concurrency control, such as device drivers, implementations of synchronization protocols, and concurrent data structures. We have implemented our new method in a software tool and demonstrated its effectiveness in diagnosing bugs in multithreaded C programs.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Program; D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Concurrency; bounded model checking; error diagnosis; program repair; unsatisfiability core; partial MAX-SAT; binate covering

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 its complexity, it is often challenging for programmers to reason about the behavior of their code. Testing is also difficult because the program execution is inherently non-deterministic. Furthermore, even after a bug is detected the programmer still needs to sift through the relevant code and the failing execution to localize the root cause. Finally, coming up with a correct repair is a non-trivial task. For example, a race condition may be eliminated either by introducing a critical section or by imposing a certain execution order via signal–wait primitives. However, it may be difficult to decide which approach is better or if a certain fix is bug free. Therefore, having a software tool to help identify the potential root cause and suggest possible repairs can be beneficial to programmers.

ISSTA'15, July 12–17, 2015, Baltimore, MD, USA
© 2015 ACM. 978-1-4503-3620-8/15/07...$15.00
http://dx.doi.org/10.1145/2771783.2771798

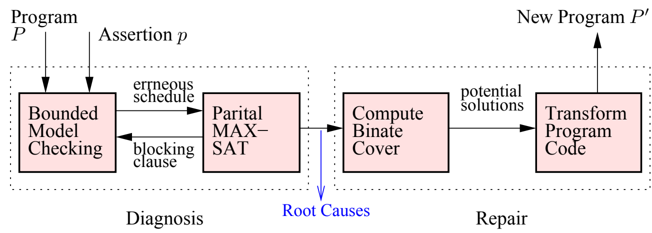


Figure 1: The *diagnose-and-repair* flow of ConcBugAssist.

Our work is inspired by recent developments in constraint-based methods for diagnosing software bugs [32, 18, 50, 56, 69, 68]. A representative of these methods is Bug-Assist [32], which uses a bounded model checker to 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], weakest preconditions [56, 69, 68] and interpolants [50], 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.

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 [32, 18, 50], 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, and then compute possible ways of repairing the source code of the program to eliminate the bug.

Figure 1 shows the overall flow of our 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, which 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 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 as input to a follow-up procedure for computing the potential repairs. By potential repair we mean modifications to the source code of the original program that are sufficient to eliminate the observed violations. As shown on the right-hand side of Figure 1, we formulate the computation of potential repairs as an instance of another constraint solving problem, i.e., the binate covering problem.

It is important to note that, since it is impossible in general, we do not attempt to fully automate the repair process by taking programmers out of the loop. Instead, we aim at leveraging program analysis techniques as a debugging aid to provide meaningful suggestions. There are three reasons for us 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 [59]. Third, in practice, developers are generally skeptical about tools that modify software code without going through the standard process of code review and certification.

We have implemented our method in a software tool based on CBMC [35] and a partial MAX-SAT solver called MSUnCore [47]. We have evaluated it on a set of multithreaded C programs. Our experimental results show that the new method is effective both in localizing the root cause of a concurrency bug and in computing potential repairs. Specifically, in all benchmark programs, the repairs suggested by our tool is consistent with the correct bug fixes as confirmed by our manual code inspection.

To summarize, this paper 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 propose a new method for computing potential repairs, by iteratively adding inter-thread ordering constraints to the program to eliminate erroneous schedules.
- We implement the new *diagnose-and-repair* framework in a software tool and demonstrate its effectiveness on a set of multithreaded C programs.

The remainder of this paper is organized as follows. First, we establish notation and review the basics of model checking concurrent programs in Section 2. Then, we present our new diagnosis method in Section 3. We present our new method for computing potential repairs in Section 4. We present the results of our experimental evaluation in Section 5. We review related work in Section 6 and finally give our conclusions in Section 7.

2. PRELIMINARIES

2.1 Bounded Model Checking (BMC)

Bounded model checking is a method for checking temporal logic properties in a state transition system by encoding the possible program executions as logical formulas and then solving them using constraint solvers. For directly analyzing software code, tools such as CBMC [35] 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. As such, 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 BMC 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. 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. For a comprehensive review, refer to [1, 35].

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.

2.2 Modeling Concurrent Programs

For ease of comprehension, we use the program in Figure 2 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 $TF_1 \wedge TF_2 \wedge Ord$,

```
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: Motivating example.

where $TF_i, i \in \{ 1, 2 \}$, 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 program execution that violates the assertion. In general the satisfying assignment consists of two types of information: a set of concrete values for the program (data) input variables, and a set of concrete values for the *clock* variables, representing the erroneous thread schedule. In the running example in Figure 2, 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, one erroneous schedule is $1 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 10$. 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. The partial MAX-SAT problem is a further extension that separates the clauses into *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 and the *minimally unsatisfiable subformula* (MUS) [38]. 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. [38] show that MUS can be computed by leveraging existing SAT and MAX-SAT solvers [49, 37, 47]. 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.

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

3. DIAGNOSING CONCURRENCY BUGS

As shown in Figure 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 used as input to compute potential bug fixes (Section 4).

3.1 Generating the Failing Executions

The first step of the diagnosis phase, 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 [35] 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. If this combined formula is satisfiable (Line 3), then there exists a data input and thread schedule ((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. 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).

To eliminate the entire set 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. 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 $\phi_\triangle$ contains the diagnosis information of all erroneous thread schedules, one (non-negated) ϕ_{core} per schedule, seen so far. In the end, $\phi_\triangle$ contains the diagnosis information across all buggy schedules.

3.3 Diagnosing the Running Example

Consider the example in Figure 2, where the first failing execution corresponds to the line numbers: $1 \to 7 \to 8 \to 9 \to 3 \to 4 \to 5 \to 10$. 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 $i \in \{1, 2, \dots\}$, is the *clock* variable associated with the instruction at line i. Let $e_i \to e_j$ 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:

$$\phi_{sch} \equiv (e_1 \to e_7) \wedge (e_7 \to e_8) \wedge \dots$$

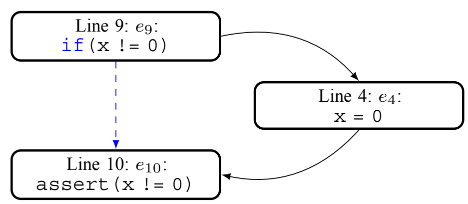
However, many of these ordering constraints are not relevant to the root cause of the error. To localize the root cause, we construct an intentionally unsatisfiable formula as follows:

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

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, $\phi_{in} \equiv$ `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.

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

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 a large set of erroneous schedules, all of which share the same core constraints in ϕ_{core}. Although this particular example requires only one iteration in Algorithm 1, in general, our diagnosis procedure needs multiple iterations to eliminate all erroneous executions. Within each iteration, we conjoin $\neg\phi_{core}$ with ϕ. At the same time, we record ϕ_{core} in $\phi_\triangle$ for latter use. When the model checker can no longer find failing executions, $\phi_\triangle$ contains the set of constraints sufficient for explaining all failing executions.

4. COMPUTING POTENTIAL REPAIRS

In this section, we present our method for computing repairs that can be presented to programmers for review and confirmation. The pseudocode of the procedure is shown in Algorithm 2, which takes the program P and the set $\phi_\triangle$ computed in the diagnosis phase as input, and returns a set $\mathcal{P}$ of new programs as output.

The procedure consists of the following steps: For each erroneous thread schedule ϕ_{sch} (and more specifically ϕ_{core}), we construct a *kill-set*, defined as the set of inter-thread ordering constraints such that if any were enforced in the program, the erroneous thread schedule would be infeasible.

Based on the *kill-sets*, we formulate the repair computation as a binate covering problem (BCP), where each repair is a *cover*, containing at least one constraint from each *kill-set*. Furthermore, these chosen constraints must not contradict with each other, or with the hard constraints that model the program logic. Since in general there may be multiple solutions to the BCP, we explore the solution

Algorithm 2 Computing the potential repairs.

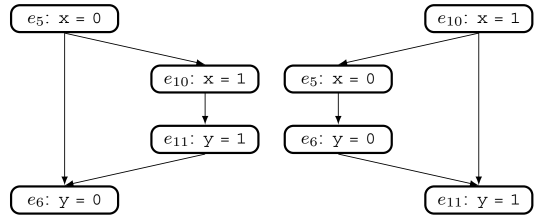
Input: Program P, and the set ϕ_Δ
Output: Set $\mathcal{P}$ of repaired programs

1: $\mathcal{P} \leftarrow \varnothing$
2: $S_{kill} \leftarrow$ ConstructKillSets(P, ϕ_Δ)
3: $S_{repair} \leftarrow$ ComputeBinateCovers(P, S_{kill})
4: **for all** $repair \in S_{repair}$ **do**
5: $\quad P' \leftarrow$ TransformProgram$(P, repair)$
6: $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{ P' \}$
7: **end for**
8: **return** $\mathcal{P}$

```
int x = 0;
int y = 0;

void f1(void) {
  x = 0;
  y = 0;
}

void f2(void) {
  x = 1;
  y = 1;
}

```

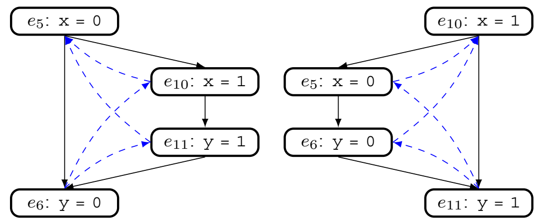
```


int main() {
  pthread_t t1, t2;
  thread_create(t1,f1);
  thread_create(t2,f2);

  thread_join(t1);
  thread_join(t2;
  assert(x == y);
  return 0;
}

```

Figure 3: Buggy program: there are atomicity violations between the two threads.



(a) (b)

Figure 4: Graphical representation of the two buggy interleavings of Figure 3. Each results in an assertion violation.



(a) (b)

Figure 5: Potential happens-before edges that block the two buggy interleavings in Figure 3. The solid edges are the ordering of erroneous interleaving while the dashed edges are happens-before edges that block the erroneous interleaving.

space to find the most efficient repairs, either in terms of the size of the code changes in the repair or its permissiveness.

Finally, we realize the chosen repairs as a modification to the original program, by enforcing the inter-thread ordering constraints using synchronization primitives such as locks, signal/wait, or the *atomic* keyword in transactional memory systems.

4.1 Constructing the Kill-Sets

First, we construct the *kill-set* for each erroneous thread schedule (an item in ϕ_Δ). The kill-set is a set of all inter-thread ordering constraints such that each constraint, when added to the original program, would be sufficient to eliminate the erroneous schedule.

Consider an example kill-set for the program in Figure 3: the two threads, `t1` and `t2`, share variables `x` and `y`. The assertion condition `(x == y)` indicates that the assignment statements in both threads should run atomically without interference from the other thread. However, this atomicity property is not enforced in either thread: `t1` can interleave in between `t2`'s updates and vice versa. As a result, there are two sets 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 in the previous section, would be able to return the localized constraints for both sets (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 $e_i \rightarrow e_j$ appears in ϕ_{core}, it means the happens-before edge is necessary for explaining why the assertion is violated.

To compute the kill-sets for ϕ_{core_1} and ϕ_{core_2} as required by Algorithm 2, 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 4 shows the graphical representations of these two erroneous thread schedules side by side. Specifically, Figure 4 (a) 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 4 (b) shows a similar schedule where the final state results in `x == 0` and `y == 1`.

Next, we compute a set of new happens-before constraints such that enforcing any of them in the original program is sufficient to prevent the erroneous interleaving (the kill-set). We define a new happens-before relation ($\rightarrow_s$) 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 4 (a), the kill-set can be constructed by adding 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 4 (a), there is an edge $e_5 \rightarrow e_{10}$. Thus, we can create a cycle 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 5 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. 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 our graph based algorithm.

Also, although any edge from a kill-set of a schedule is sufficient for eliminating the schedule, sometimes, edges chosen from different kill-sets contradict each other. For example, the erroneous schedule in Figure 5 (a) can be eliminated with the insertion of $e_6 \rightarrow_s e_{11}$ while the one in Figure 5 (b) can be eliminated with the insertion of $e_{11} \rightarrow_s e_6$. However, these two happens-before edges cannot simultaneously be enforced in the program. In the remainder of this section, we formulate the repair computation as a binate covering problem, which ensures that the solution is free of such contradictions.

4.2 Computing the Binate Cover

The repair computation, in general, can be formulated as a binate covering problem (BCP) [64]. BCP has been studied extensively in logic synthesis and combinatorial optimization. Here, our goal is to find a valid set of happens-before edges, at least one from each kill-set (all kill-sets are covered) without introducing any contradiction.

Let the set of happens-before constraints in the union of all kill-sets be represented by $S = \{s_1, \ldots, s_n\}$ and the cost of selecting each happens-before constraint s_i is k_i, where $k_i \geq 0$. We associate a Boolean variable x_i to s_i, which has a value 1 if s_i is selected and 0 otherwise. The binate covering problem can be defined as finding a subset $C \subseteq S$ (or cover) that minimizes $\Sigma_{i=1}^{n} k_i x_i$ subject to a Boolean constraint $A(x_1, x_2, \ldots, x_n)$, where A precisely specifies the set of valid solutions.

In our application, the constraint function A is a conjunction of two parts. The first part is $\bigwedge_i^n KS_i$, where each KS_i represents that at least one constraint from the kill-set of schedule i must be chosen. The second part, which we refer to as ω, is a constraint that specifies the compatibility of all the chosen constraints based on their definitions as well as the semantics of the concurrent program e.g., the chosen happens-before edge cannot violate a happens-before edge already existing in the program.

For example, if we use x_1 to denote the selection of the edge $e_{10} \rightarrow_s e_5$ and use x_2 to denote the selection of the edge $e_5 \rightarrow_s e_{10}$, we need to add the Boolean constraint $(\neg x_1 \vee \neg x_2)$ to ω since, by definition, these two variables cannot be set to 1 simultaneously. If we use x_3 to denote the selection of the edge $e_6 \rightarrow_s e_{10}$, we also need to add the Boolean constraint $(\neg x_1 \vee \neg x_3)$ to ω, because these two edges would from a cycle with the program-order constraint $e_5 \rightarrow e_6$ (which is always true).

To make the example complete, we now show the two kill-sets for the program in Figure 5. The first kill-set is defined as follows:

$$KS_1 = (e_{10} \rightarrow_s e_5) \vee (e_{11} \rightarrow_s e_5) \vee (e_6 \rightarrow_s e_{10}) \vee (e_6 \rightarrow_s e_{11})$$

Enforcing any of these new happens-before edges in the program would be sufficient for blocking the erroneous thread schedule. Similarly, the second kill-set is defined as follows:

$$KS_2 = (e_5 \rightarrow_s e_{10}) \vee (e_6 \rightarrow_s e_{10}) \vee (e_{11} \rightarrow_s e_5) \vee (e_{11} \rightarrow_s e_6)$$

Finally, a valid repair (for blocking all erroneous interleavings) is a satisfiable assignment to the formula $A = KS_1 \wedge KS_2 \wedge \omega$.

When the constraint formula A is given in a product-of-sums form, one can represent the BCP using a table, where each variable in A (a happens-before edge) is a column and each clause (sum) is a row, and the problem can be interpreted as one of finding a subset C of the columns of minimum cost, such that for every row is covered. The binate covering problem is NP-hard, but in practice, can also be solved by efficient branch-and-bound algorithms [64].

As shown in Figure 6, for our example from Figure 3, there are a total of sixteen possible solutions, among which there are four valid (unique) solutions:

- Solution (A): $e_6 \rightarrow_s e_{10}$
- Solution (B): $e_{11} \rightarrow_s e_5$
- Solution (C): $e_5 \rightarrow_s e_{10} \wedge e_6 \rightarrow_s e_{11}$
- Solution (D): $e_{10} \rightarrow_s e_5 \wedge e_{11} \rightarrow_s e_6$

All the other solutions are either invalid, meaning that they lead to cycles in the graph, or are equivalent to one of these four solutions.

Due to the use of compatibility constraints (ω and A) in BCP, our method guarantees that the repair will not introduce certain type of deadlocks, i.e., the ones caused by incompatibility of newly added happens-before edges and the original thread program order constraints. However, it is possible for a repair to introduce other type of deadlocks, e.g., from reversed lock orderings between threads. Since our method uses bounded model checking as the underlying verification procedure, in principle, we cannot guarantee that the repair is always correct. A possible remedy for the deadlock problem is to verify the suggested repairs using a static deadlock analysis and then filter out the erroneous repairs.

No.	Cover Set	Valid	Simplified Result
1	$(e_{10} \rightarrow e_5) \wedge (e_5 \rightarrow e_{10})$	cycle	
2	$(e_{10} \rightarrow e_5) \wedge (e_{11} \rightarrow e_5)$	YES	$(e_{11} \rightarrow e_5)$
3	$(e_{10} \rightarrow e_5) \wedge (e_6 \rightarrow e_{10})$	cycle	
4	$(e_{10} \rightarrow e_5) \wedge (e_{11} \rightarrow e_6)$	YES	$(e_{10} \rightarrow e_5) \wedge (e_{11} \rightarrow e_6)$
5	$(e_{11} \rightarrow e_5) \wedge (e_5 \rightarrow e_{10})$	cycle	
6	$(e_{11} \rightarrow e_5) \wedge (e_{11} \rightarrow e_5)$	YES	$(e_{11} \rightarrow e_5)$
7	$(e_{11} \rightarrow e_5) \wedge (e_6 \rightarrow e_{10})$	cycle	
8	$(e_{11} \rightarrow e_5) \wedge (e_{11} \rightarrow e_6)$	YES	$(e_{11} \rightarrow e_5)$
9	$(e_6 \rightarrow e_{10}) \wedge (e_5 \rightarrow e_{10})$	YES	$(e_6 \rightarrow e_{10})$
10	$(e_6 \rightarrow e_{10}) \wedge (e_{11} \rightarrow e_5)$	cycle	
11	$(e_6 \rightarrow e_{10}) \wedge (e_6 \rightarrow e_{10})$	YES	$(e_6 \rightarrow e_{10})$
12	$(e_6 \rightarrow e_{10}) \wedge (e_{11} \rightarrow e_6)$	cycle	
13	$(e_6 \rightarrow e_{11}) \wedge (e_5 \rightarrow e_{10})$	YES	$(e_6 \rightarrow e_{11}) \wedge (e_5 \rightarrow e_{10})$
14	$(e_6 \rightarrow e_{11}) \wedge (e_{11} \rightarrow e_5)$	cycle	
15	$(e_6 \rightarrow e_{11}) \wedge (e_6 \rightarrow e_{10})$	YES	$(e_6 \rightarrow e_{10})$
16	$(e_6 \rightarrow e_{11}) \wedge (e_{11} \rightarrow e_6)$	cycle	

Figure 6: Binate Covering: The set of all valid repairs.

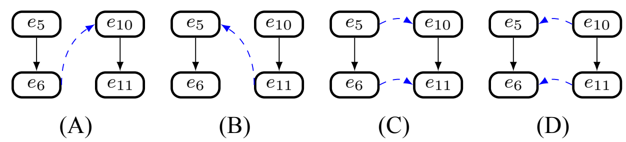


Figure 7: Happens-before constraints added by the solutions.

In general, repairs computed in this section are merely suggestions to the programmer, who is expected to review the solution and ultimately decide if a repair should be applied. We do not attempt to fully automate this process, since in the absence of a complete formal specification of the intended program behavior, the debugging process cannot be completely automated. Nevertheless, we shall show in the experiments section that the repairs suggested by our tool are often the correct repairs and in many cases are optimal in terms of the size of code changes and/or the permissiveness.

4.3 Realizing the Solution

Just like the four solutions computed above, in general, valid solutions to the BCP form a hierarchy. For a closer look at the different thread ordering enforced by these solutions, see the scenarios illustrated in Figure 7. Here, the dashed edges are newly added happens-before constraints to the program while the solid edges are those enforced by the program order. It is clear that any of these four solutions would be sufficient for repairing the program. However, they also have different cost in terms of both ease of implementation and performance overhead.

One way to rank these solutions is using the implementation cost. For example, to enforce $e_{11} \rightarrow_s e_5$ in the program, a *cond-wait* can be inserted before e_5 and a *cond-signal* inserted after e_{11}. If we define the implementation cost as the number of signal–wait pairs added to the program code, Solution (A) and Solution (B) would be better than Solution (C) and Solution (D).

Another way to rank these solutions is using *permissiveness*, i.e., the number of allowed interleavings. In this case, Solution (A) and Solution (B) would be worse than Solution (C) and Solution (D). We say Solution (A) is less permissive than Solution (C) because it can eliminate all interleavings that are eliminated by Solution (C), and more. If the goal is to allow the program more freedom to "choose" thread schedules in the hope that it leads to better performance, Solution (C) and solution (D) are better choices.

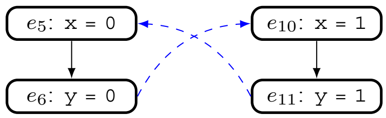


Figure 8: Graphical representation of the *either-or* edges—Solutions (a) and (b)—for fixing the bug in Figure 3. Blue edges are happens-before relations returned by the bug repair procedure. Black edges are intra-thread program-order relations.

Interestingly, there are *composite* solutions, a combination of multiple elementary solutions, that get us the best of both worlds. One such composite solution is enforcing either Solution (A) or Solution (B) at runtime. This composite approach can be realized by inserting a mutex lock–unlock pair to surround lines 5–6 and lines 10–11, to make them mutually exclusive. While the previous elementary solutions (in Figure 7) require `t1` to always happen before `t2` (or vice versa), this composite solution, however, allows for the program have either behavior. Such a composite solution is still bug free and allows greater concurrency. However, it remains to be shown if such composite solutions can be identified automatically.

In our method, we first systematically search for the elementary solutions while minimizing the implementation cost, and then try to combine them together to increase the permissiveness. Toward this end, we examine the set of all happen-before edges that forms a valid repair. As an example, we consider the combination of solutions (A) and (B): $e_{11} \rightarrow_s e_5$, or $e_6 \rightarrow_s e_{10}$. Note that there is an implicit happens-before edge, or program-order constraint, between the two assignments within a thread. That is, $e_5 \rightarrow e_6$ and similarly $e_{10} \rightarrow e_{11}$ (since they are within the same thread) is fixed. We assume that the programs are sequentially consistent. As a result, the two happens-before edges specify that either $(e_{10} \rightarrow e_{11}) \rightarrow_s (e_5 \rightarrow e_6)$, or $(e_5 \rightarrow e_6) \rightarrow_s (e_{10} \rightarrow e_{11})$. This is represented graphically in Figure 8.

For repairs with more than two possible solutions, we identify this situation by building a graph such as in Figure 8 with intra-thread happens-before edges for pairs of possible solutions such that they do not contradict each other. Then, we group statements from the same thread (e_{10} and e_{11}, and e_5 and e_6 in this example) together. If the result is graph with two threads connected by two *either-or* edges to from a cycle, then we can insert a mutex lock/unlock pair before/after the intra-thread statements. Otherwise, we select one of the satisfying happens-before edges and enforce it by inserting a condition variable signal/wait pair.

The critical sections computed above for the composite solution do not have to be enforced by adding lock-unlock pairs. Another way to implement such repair is to use the *atomic* keyword in a transactional memory system.

5. EXPERIMENTS

We have implemented our diagnosis and repair methods in a software tool called *ConcBugAssist* based on the latest version of the CBMC [35] model checker, which supports the verification of multithreaded programs [1]. We used the MSUnCore [47] partial MAX-SAT solver during the diagnosis and repair computation.

We have evaluated our methods on 34 benchmark programs. Our experiments were designed to answer two research questions:

- Can our diagnosis method accurately localize the root cause of a concurrency bug?
- Can our repair method compute meaningful code modifications to eliminate the bug?

Table 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 bug. The last column also shows the origin. Our benchmarks can be classified into four groups.

Table 1: Characteristics of the benchmark programs.

Name	LOC	Threads	Bug Type	Origin
boop	98	3	atomicity violation	[60]
testc	19	2	order violation	[60]
fibbench	47	3	order violation	[60]
fibbench_longer	45	3	order violation	[60]
reorder	105	5	order violation	[60]
account	58	4	order violation	[60]
read_write	140	5	order violation	[60]
barrier	85	4	order violation	[60]
lazy01	55	4	data race	[60]
VectPrime02	183	3	data race	[8]
lineEq2t01	58	3	data race	[8]
linux-tg3	115	3	order violation	[8]
linux-iio	87	3	atomicity violation	[8]
mysql-169	27	3	atomicity violation	[72, 53]
mysql-12848	142	2	atomicity violation	[72, 52]
mysql-3596	83	3	order violation	[45]
mysql-644	165	3	order violation	[45]
apache-21287	79	3	atomicity violation	[2]
apache-25520	192	3	data race	[3]
freebsd-aa	104	4	order violation	[70]
cherokee-0.9.2	188	3	atomicity violation	[72]
llvm-8441	244	3	order violation	[44]
gcc-25330	86	3	atomicity violation	[22]
gcc-3584	104	3	data race	[23]
gcc-21334	94	3	data race	[21]
gcc-40518	114	3	data race	[24]
transmission-1.42	78	3	order violation	[72]
glib-512624	98	3	atomicity violation	[26]
jetty-1187	74	3	order violation	[30]
mozilla-61369	68	3	order violation	[45]
hash_table	156	3	atomicity violation	[27]
list_seq	122	3	atomicity violation	[27]
counter_seq	41	3	data race	[27]
queue_seq	97	3	data race	[27]

The first group consists of the POSIX threads related buggy programs from the 2015 Software Verification Competition [60] (SV-COMP). 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 programs used by Bloem et al. [8], where the first two are synthetic benchmarks, while *linux-iio* and *linux-tg3* are real bugs found in the industrial I/O subsystem (IIO) of the linux kernel (http://git.io/JjCEXg), and Broadcom Tigon3 (TG3) Ethernet driver (http://git.io/7wWrKw), respectively.

The third group consists of bug patterns extracted from various versions of open source applications. They are reported in MySQL [51], the Apache Web Server [4], the FreeBSD Operating System [19], the Cherokee Web Server [11], the LLVM Compiler Framework [43], the GNU Compiler Collection [20], the Linux Kernel [39], the Transmission BitTorrent client [61], the GNOME Library [25], the Jetty HTTP Server [29], and Mozilla's XPCOM library [67]. 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 Art of Multiprocessor Programming book [27]. Some of these programs are stripped off the synchronization operations intentionally to see if our method can correctly repair them back to normal.

5.1 Diagnosis Results

First, we evaluate the effectiveness of our diagnosis algorithm. Table 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 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 expect 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 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 the repair.

Table 2: Summary of the error diagnosis results.

Name	Time (s)	Iter.	Constr./ (ϕ_{sch})	Size of Root Cause: Constr./ (ϕ_{core})	Size of Root Cause: Unique Constr.	Red. Ratio
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%

Table 3: Summary of the repair computation results.

Name	Time (s)	No. of Repairs	Type of Fix: Atomic	Type of Fix: Signal–Wait
boop	0.1	5	✓	✓
testc	0.1	2	×	✓
fibbench	2.7	97	×	✓
fibbench_longer	4.2	97	×	✓
reorder	0.6	34	×	✓
account	0.3	73	×	✓
read_write	4.8	68	×	✓
barrier	0.1	33	×	✓
lazy01	0.2	54	✓	✓
VectPrime02	0.1	7	✓	✓
lineEq2t01	1.2	10	✓	✓
linux-tg3	0.1	2	×	✓
linux-iio	0.5	93	✓	✓
mysql-169	0.1	13	✓	✓
mysql-12848	0.1	27	✓	✓
mysql-3596	0.1	1	×	✓
mysql-644	0.1	2	×	✓
apache-21287	0.1	7	✓	✓
apache-25520	0.2	10	✓	✓
freebsd-aa	1.1	90	×	✓
cherokee-0.9.2	0.2	10	✓	✓
llvm-8441	2.1	95	×	✓
gcc-25330	0.1	12	✓	✓
gcc-3584	0.2	39	✓	✓
gcc-21334	0.2	5	✓	✓
gcc-40518	0.1	35	✓	✓
transmission-1.42	0.1	2	×	✓
glib-512624	0.1	27	✓	✓
jetty-1187	0.1	2	×	✓
mozilla-61369	0.1	1	×	✓
hash_table	0.6	27	✓	✓
list_seq	0.4	15	✓	✓
counter_seq	0.1	7	✓	✓
queue_seq	0.1	3	✓	✓

5.2 Repair Results

Next, we evaluate the effectiveness of the repair algorithm. Table 3 summarizes the results, where Columns 1 and 2 show the program name and the repair time, respectively. Column 3 shows the number of valid repairs returned by our method. Columns 4–5 show the types of these repairs. Specifically, if the bug can be fixed by adding critical sections, either through the insertion of lock-unlock pairs or using the *atomic* keyword, we put a ✓ in Column 4. Similarly, if the bug can be fixed by adding signal-wait pairs, we put a ✓ in Column 5.

Since the benchmarks used in our evaluation span a wide range of concurrency bugs, the results shown in Table 3 are particularly promising. In general, our repair algorithm can quickly return multiple repairs. Some of these repairs rely on the insertion of atomic blocks, some rely on the insertion of signal-wait pairs, and some may be fixed using both approaches.

Currently, our tool ranks the repairs before presenting them to the user. For elementary solutions, the ranking is based on the number of happens-before constraints used in the solutions (fewer is better). In addition, we always search for composite solutions that combine multiple elementary solutions to allow for greater concurrency, and rank them higher. We leave the design and analysis of more complex ranking systems as future work.

Our repair procedure returns a surprisingly large number of repairs for certain programs. We believe it is due to the many distinct but semantically equivalent repairs in these programs. For example, in the buggy list implementation in Figure 9, executing Line 11 before Line 19 is a different solution than running Line 12 before

Table 4: Detailed statistics of the repair computation.

	Repairs			Kill-Set Size	
Name	Elementary	Composite	Avg. Size	Avg.	Total
boop	4	1	1	4	4
testc	2	0	1	2	2
fibbench	97	0	2	35	70
fibbench_longer	97	0	2	48	96
reorder	34	0	5	6	87
account	73	0	3	12	36
read_write	68	0	3	76	2149
barrier	33	0	6	6	57
lazy01	53	1	2	8	16
VectPrime02	6	1	2	3	6
lineEq2t01	94	6	2	22	44
linux-tg3	2	0	1	2	2
linux-iio	87	6	3	5	25
mysql-169	12	1	2	4	8
mysql-12848	26	1	3	4	16
mysql-3596	1	0	1	1	1
mysql-644	2	0	1	2	2
apache-21287	6	1	2	3	6
apache-25520	9	1	3	7	112
freebsd-aa	90	0	5	21	1025
cherokee-0.9.2	9	1	3	5	52
llvm-8441	95	0	3	29	612
gcc-25330	11	1	2	4	8
gcc-3584	38	1	3	6	22
gcc-21334	4	1	1	4	4
gcc-40518	31	4	2	7	14
transmission-1.42	2	0	2	2	3
glib-512624	26	1	3	7	116
jetty-1187	2	0	1	2	4
mozilla-61369	1	0	1	1	1
hash_table	26	1	3	9	400
list_seq	14	1	3	5	95
counter_seq	6	1	2	3	6
queue_seq	2	1	2	2	4

Line 19. However, the semantics of both fixes are equivalent. This can lead to a large number of potential combinations, especially for those programs which require multiple happens-before edges to be fixed (e.g., $11 \rightarrow 19$ can be substituted with $12 \rightarrow 19$ and vice versa). However, our repair procedure automatically ranks solutions based on size so the user can find a suitable repair without having to examine all repairs.

Detailed statistics of binate cover computation are in Table 4, which breaks down the set of repairs into *elementary* and *composite* repairs, and show their numbers in Columns 2 and 3. We also show in Column 4 the average size of a repair, in terms of the number of happens-before ordering constraints it contains. Columns 5 and 6 shows the average size of the kill-set and the total number of happens-before constraints in all kill-sets (Section 4.1).

Overall, the repair computation has little overhead compared to the diagnosis procedure. This is likely due to the small size of the kill-sets compared to the size of the model checking formula. Although the binate covering problem has exponential complexity in the worst case, the relatively small size of the kill-sets makes it fast. We expect the run time of BCP solving to increase significantly as the kill-sets get larger. In such cases, approximate algorithms can be used to quickly compute a suboptimal solution in order to scale up the algorithm.

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) used concurrently by multiple threads. A shortened version of its source code can be seen in Figure 9. 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.

```
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 9: Buggy *list_seq* with two potential repairs (s_1 and s_2).

The bug is due to lack of atomicity in `list_add()`: the insertion of an item (Line 7) is not atomic with the update of the lists size (Line 8). Our method 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 9 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 an *either-or* solution: either thread 1 can go first or thread 2 can go first. The highest ranked solution in our repair procedure is to enforce this *either-or* edges: the calls to `list_add` are surrounded with calls to `mutex_lock` and `mutex_unlock` to enforce atomicity of the operation. 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 10. 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.

The highest ranked solution found by our repair algorithm is shown as the arrow from Line 2 to Line 5 in the figure. That is, the bug can be prevented by enforcing that Line 2 is always executed before Line 5. Our repair algorithm also return another solu-

```
void t1_main() {
  bandwidth = malloc(...)
}
void t2_main() {
  assert(bandwidth != NULL);
  *bandwidth = ...
}
int main() {
  thread_t t1, t2;
  thread_create(&t1, t1_main);
  thread_create(&t2, t2_main);
  thread_exit();
}

```

Figure 10: Relevant portion of the code in *transmission-1.42*.

tion, which has two happens-before edges and therefore is inferior to the first solution. Furthermore, our repair algorithm shows that there are no composite solution that can combine the elementary solutions together. Therefore, the repair shown in Figure 10 will be reported 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.

6. RELATED WORK

Our work was inspired by the set of recent works on using constraint solvers for diagnosing software bugs [32, 18, 50, 56, 69, 68]. However, these methods were designed solely for diagnosing logical bugs in sequential software. Furthermore, the type of repairs computed by some of these methods were fairly limited. For example, BugAssist [32] focuses on repairs that are minor mutations of existing expressions in the program, e.g., replacing `if(c > 0)` with `if(c >= 0)` as in the *off-by-one* bug pattern. None of these methods handle concurrency bugs in multithreaded programs. In contrast, our work focuses primarily on diagnosing concurrency bugs and suggesting repairs.

Our work is related to methods for synthesizing synchronizations among concurrent threads based on a specification [63, 7] or by making interleaved executions conform to sequential executions [8]. For example, the method proposed by Bloem et al. [8] 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. [65, 66] 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 et al. [42, 40] extended the approach to include certain types of atomicity violations. Liu et al. [41] also proposed a method for fixing linearizability violations in concurrent data structures. However, these violations are still not general concurrency bugs targeted by our method, which include any non-deadlock concurrency bug that can be modeled as violation of an assertion.

Krena et al. [34, 36, 33] and Jin et al. [31] 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 [12, 73, 28, 54, 71, 6], but cannot systematically detect failing executions. Furthermore, they tend to focus more on helping the user diagnose bugs manually as opposed to computing repairs.

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.

Our work is also related, at the high level, to the theoretical work on program synthesis [16, 46, 55, 15, 9] and controller synthesis [57, 58], where the focus is on automated construction of systems from their specifications. For concurrent software, there are also methods for automated lock insertion and refinement [5, 48, 62, 17, 10, 74], which assume the critical sections or correctness specifications are provided as input and their goal is adding locks or delays to transparently ensure such properties. These methods differ from our approach in that they assume the availability and correctness of a complete specification or golden model, which we do not have.

Our method relies on CBMC as the underlying verification procedure, which can limit its ability of handling large programs. The scalability problem may be addressed in two ways. First, the Boolean SAT solvers used in the diagnosis phase may be replaced by SMT solvers [14], which tend to work on higher levels of abstractions and therefore are potentially more scalable than Boolean SAT solvers. Second, our diagnosis and repair methods may be applied to a Boolean abstraction of the program, created using well-known predicate abstraction tools such as SATABS [13], as opposed to the concrete program—it may result in some precision loss due to the use of predicate abstraction, but at the same time will significantly boost the runtime performance. We leave the exploration of such optimizations for future work.

7. CONCLUSIONS

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 presented a constraint based method for computing potential program repairs by iteratively adding additional happens-before constraints to block the erroneous thread schedules. These new methods have been implemented in a software tool and evaluated on a set of multithreaded C programs. Our experiments show that the proposed methods are effective in explaining concurrency bugs and suggesting meaningful repairs.

8. ACKNOWLEDGMENTS

This work was primarily supported by the NSF under grants CCF-1149454, CCF-1405697, and CCF-1500024. Partial support was provided by the ONR under grant N00014-13-1-0527. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

9. REFERENCES

[1] Jade Alglave, Daniel Kroening, and Michael 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] Paul C. Attie. Synthesis of large concurrent programs via pairwise composition. In *International Conference on Concurrency Theory*, pages 130–145, 1999.

[6] Mitra Tabaei Befrouei, Chao Wang, and Georg Weissenbacher. Abstraction and mining of traces to explain concurrency bugs. In *International Conference on Runtime Verification*, pages 162–177, 2014.

[7] Roderick Bloem, Krishnendu Chatterjee, Swen Jacobs, and Robert Könighofer. Assume-guarantee synthesis for concurrent reactive programs with partial information. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 517–532, 2015.

[8] Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Ausserlechner, and Raphael Spork. Synthesis of synchronization using uninterpreted functions. In *International Conference on Formal Methods in Computer-Aided Design*, pages 35–42, 2014.

[9] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield synthesis: Runtime enforcement for reactive systems. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 533–548, 2015.

[10] Sigmund Cherem, Trishul M. Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 304–315, 2008.

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

[12] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *International Symposium on Software Testing and Analysis*, pages 210–220, 2002.

[13] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.

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

[15] Jyotirmoy V. Deshmukh, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Logical concurrency control from sequential proofs. In *European Symposium on Programming*, pages 226–245, 2010.

[16] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[17] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 291–296, 2007.

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

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

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

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

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

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

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

[25] GLib reference manual URL: https://bugzilla.gnome.org/show_bug.cgi? id=512624.

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

[27] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[28] Nicholas Jalbert and Koushik Sen. A trace simplification technique for effective debugging of concurrent programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 57–66, 2010.

[29] Jetty servlet engine URL: http://www.eclipse.org/jetty/.

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

[31] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 389–400, 2011.

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

[33] Bohuslav Krena, Zdenek Letko, Yarden Nir-Buchbinder, Rachel Tzoref-Brill, Shmuel Ur, and Tomás 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.

[34] Bohuslav Krena, Zdenek Letko, Rachel Tzoref, Shmuel Ur, and Tomás Vojnar. Healing data races on-the-fly. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pages 54–64. ACM, 2007.

[35] Daniel Kroening and Michael Tautschnig. CBMC—C bounded model checker. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 389–391, 2014.

[36] Zdenek Letko, Tomás Vojnar, and Bohuslav 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.

[37] Chu-Min Li, Zhiwen Fang, and Ke 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.

[38] Mark H. Liffiton, Zaher S. Andraus, and Karem A. Sakallah. From Max-SAT to Min-UNSAT: Insights and applications. Technical Report CSE-TR-506-05, University of Michigan, 2005.

[39] The Linux kernel archives URL: http://kernel.org.

[40] Peng Liu, Omer Tripp, and Charles Zhang. Grail: context-aware fixing of concurrency bugs. In *ACM SIGSOFT*

Symposium on Foundations of Software Engineering, pages 318–329, 2014.

[41] Peng Liu, Omer Tripp, and Xiangyu Zhang. Flint: fixing linearizability violations. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 543–560, 2014.

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

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

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

[45] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan 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.

[46] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, 1984.

[47] Joao Marques-Silva. The MSUnCore MAXSAT Solver.

[48] Bill McCloskey, Feng Zhou, David Gay, and Eric A. Brewer. Autolocker: synchronization inference for atomic sections. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 346–358, 2006.

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

[50] Vijayaraghavan Murali, Nishant Sinha, Emina Torlak, and Satish Chandra. What gives? A hybrid algorithm for error trace explanation. In *International Conference on Verified Software*, pages 270–286, 2014.

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

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

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

[54] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: fault localization in concurrent programs. In *International Conference on Software Engineering*, pages 245–254, 2010.

[55] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 179–190, 1989.

[56] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. DARWIN: an approach to debugging evolving programs. *ACM Trans. Softw. Eng. Methodol.*, 21(3):19, 2012.

[57] R. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*, 25(1):206–230, 1987.

[58] R. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proc. of the IEEE*, pages 81–98, 1989.

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

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

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

[62] Mandana Vaziri, Frank Tip, and Julian 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.

[63] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 327–338, 2010.

[64] T. Villa, T. Kam, R.K. Brayton, and A.L. Sangiovanni-Vincenteili. Explicit and implicit algorithms for binate covering problems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(7):677–691, Jul 1997.

[65] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 281–294, 2008.

[66] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott A. Mahlke. The theory of deadlock avoidance via discrete control. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 252–263, 2009.

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

[68] Qiuping Yi, Zijiang Yang, Jian Liu, Chen Zhao, and Chao Wang. Explaining software failures by cascade fault localization. *ACM Transactions on Design Automation of Electronic Systems*, 2015.

[69] Qiuping Yi, Zijiang Yang, Jian Liu, Chen Zhao, and Chao Wang. A synergistic analysis method for explaining failed regression tests. In *International Conference on Software Engineering*, 2015.

[70] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 26–36, 2011.

[71] Elad Yom-Tov, Rachel Tzoref, Shmuel Ur, and Shlomo 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.

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

[73] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, 2010.

[74] Lu Zhang and Chao Wang. Runtime prevention of concurrency related type-state violations in multithreaded applications. In *International Symposium on Software Testing and Analysis*, pages 1–12, 2014.