Concurrency Debugging with MaxSMT

Miguel Terra-Neves[1] and **Nuno Machado**[2] and **Inês Lynce**[1] and **Vasco Manquinho**[1]
[1] INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
{neves, ines, vmm}@sat.inesc-id.pt
[2] Teradata Spain and HASLab - INESC TEC / Universidade do Minho, Portugal
nuno.machado@teradata.com

Abstract

Current Maximum Satisfiability (MaxSAT) algorithms based on successive calls to a powerful Satisfiability (SAT) solver are now able to solve real-world instances in many application domains. Moreover, replacing the SAT solver with a Satisfiability Modulo Theories (SMT) solver enables effective MaxSMT algorithms. However, MaxSMT has seldom been used in debugging multi-threaded software.

Multi-threaded programs are usually non-deterministic due to the huge number of possible thread operation schedules, which makes them much harder to debug than sequential programs. A recent approach to isolate the root cause of concurrency bugs in multi-threaded software is to produce a report that shows the differences between a failing and a non-failing execution. However, since they rely solely on heuristics, these reports can be unnecessarily large. Hence, reports may contain operations that are not relevant to the bug's occurrence.

This paper proposes the use of MaxSMT for the generation of minimal reports for multi-threaded software with concurrency bugs. The proposed techniques report situations that the existing techniques are not able to identify. Experimental results show that using MaxSMT can significantly improve the accuracy of the generated reports and, consequently, their usefulness in debugging the root cause of concurrency bugs.

1 Introduction

Nowadays, multicore and multiprocessor architectures have become the dominant platforms. To take full advantage of these computer architectures, one has to resort to parallel and concurrent programming. However, multi-threaded programs are much more challenging to develop than sequential programs, as they typically allow a huge diversity of thread operation schedules that can yield different outcomes. Among the many possible schedules, some may result in undesirable behavior, like a crash or incorrect output (Lu et al. 2008). Such *failing schedules* (i.e. schedules that cause the program to fail) are typically non-deterministic and hard to reproduce, which renders the debugging of multi-threaded programs notoriously difficult.

A concurrency bug occurs when some shared memory is incorrectly changed due to a specific schedule of operations of concurrent threads being executed. A large body of work has focused on the ability to reproduce concurrency bugs in multi-threaded programs, namely *deterministic execution* (Berger et al. 2009; Devietti et al. 2010; Olszewski, Ansel, and Amarasinghe 2009) and *deterministic record and replay* (Lee et al. 2009; Altekar and Stoica 2009; Huang, Zhang, and Dolby 2013). However, simply replicating a failing schedule does not provide any special insight into the reason why the program has failed. As a result, finding the bug in the code that allows the failure to surface can still be a daunting task.

One way to ease the diagnosis of concurrency bugs is to report the differences between a failing execution and a non-failing execution. Given a failing schedule and a failure condition, some systems try to produce a similar *non-failing* schedule by systematically reordering pairs of thread events in the failing execution. For each alternative schedule resulting from a thread event swapping, a *Satisfiability Modulo Theories (SMT)* model is built that encodes the program's logic and synchronization constraints of the original execution and a condition representing the absence of the failure. Next, an SMT solver is used to check if there is an alternate, non-failing schedule that satisfies the constraints. After finding a feasible non-failing schedule, a report is generated that highlights the differences between the original failing schedule and the non-failing. Hence, the programmer only needs to focus on the differences in order to fix the bug.

Although effective in some cases, this technique has several limitations. In particular, if more than two pairs of operations need to be reordered simultaneously in order to prevent the failure, then there is no guarantee that a non-failing schedule will be generated.

Alongside, there are systems that expose and isolate concurrency errors that depend on both the thread operation schedule and execution path. However, these systems do not guarantee that the failing and non-failing executions exhibit similar orderings of operations. As a consequence, the generated reports can be unnecessarily long and hard to analyze.

Recently, new *Maximum Satisfiability Modulo Theories* (MaxSMT) algorithms have been proposed. However, its application in many areas is still in its infancy. In this paper, we propose the usage of MaxSMT to improve the effectiveness of reports produced by systems that analyze concurrency bugs in multi-threaded programs. In particular, we argue that a *minimal* explanation for the occurrence of a con-

(initially $x = 0, y = 1, z = -1$)

T1	**T2**	**T3**	**T4**	**T5**
1. $T2$.join()	1. $x = x + y$	1. $x = x + z$	1. $y = y + 1$	1. $z = z + 1$
2. $T3$.join()				
3. $T4$.join()				
4. $T5$.join()				
5. assert($x == 0$)				

Figure 1: Schedule dependent bug in multi-thread program.

currency bug in a multi-threaded program can be obtained by maximizing the similarities between the failing schedule and non-failing schedules. Hence, the main contributions of this paper are as follows: (i) three complementary metrics for measuring schedule similarity and a MaxSMT model that optimizes these metrics in order to produce a report with a minimal explanation of a concurrency bug; (ii) a novel approximation algorithm that considerably speeds up the MaxSMT model solving procedure by exploiting domain knowledge, and (iii) an extensive experimental evaluation, with benchmark and real-world concurrency bugs, showing that the proposed model produces more accurate reports than the previous approaches based on heuristics.

The rest of the paper is organized as follows: Section 2 provides background on concurrency bugs, SMT and other concepts necessary for this work. Section 3 exemplifies limitations of current systems for concurrency debugging. Section 4 reviews MaxSMT and describes the model for obtaining minimal reports. Section 5 evaluates the contributions of the paper and discusses the results of the experiments. Finally, Section 6 concludes the paper.

2 Background

This section defines the concepts used in the remainder of the paper. First, we review concurrency bugs. Next, we introduce the Satisfiability Modulo Theories formalism. Finally, two types of reports on concurrency bugs are described.

2.1 Concurrency Bugs

A concurrency bug is an error in multi-threaded code that may cause the program to fail depending on the order in which thread operations are executed. An operation is either a read/write of a variable or a synchronization primitive (e.g. lock, unlock, fork, etc). As an example, consider the multi-threaded program with five threads (T1–T5) shown in Figure 1. The program has three shared variables: x, y and z. Variable x is accessed by T1, T2 and T3, y is accessed by T2 and T4, and z by T3 and T5. T1 simply waits for all other threads to terminate and then checks if x is 0.

This program can fail under two different scenarios, as the assertion $x == 0$ is violated when T4 executes before T2 or T5 before T3, but not otherwise. For instance, a failing schedule can occur if the sequence of execution is: <T4, T5, T2, T3>, since the value of x is 2 when the assertion is tested. In this case, the failure is caused by two concurrency bugs, namely a data race on y between T2 and T4 and a data race on z between T3 and T5. A *data race* occurs when two different threads access the same variable without synchronization, and at least one of the accesses is a write.[1]

For this data race, T2 reads the value of y concurrently with the write on y by T4. Since the execution is nondeterministic, one may observe a schedule where T4 increments y before T2 adding y to x, thus causing x to be greater than 0 at the end of the program. However, if T2 increments x prior to T4 setting y to 2 (and T5 also executes before T3 setting z to 0), then $x = 0$ at the end of the execution and the program satisfies the assertion. In this example, each thread executes the exact same sequence of operations in both the failing and non-failing executions. Hence, since the difference between the executions concerns only the thread interleaving, the bug is said to be strictly *schedule dependent*.

However, not all concurrency bugs are strictly schedule-dependent. For example, consider the multi-threaded program in Figure 2, which has three threads (T1-T3) and four shared variables (x, y, z and w). The program will fail if x is not greater than 0 at the end of T1's execution.

As depicted in Figure 3, it may be the case that T2 runs before T1, causing w to be incremented at line 2 of T1, and T3 sets $y = 0$ in-between lines 4 and 5 of T1, i.e. after T1 increments w but before checking if $y == 0$. This interleaving will cause x to be decremented to 0 at line 6 of T1 and, consequently, violate the assertion at line 7. Notice that the failing and non-failing executions of this program necessarily differ in their control-flow paths. As such, the bug is deemed *path-schedule dependent*.

2.2 Satisfiability Modulo Theories

A propositional formula ϕ in Conjunctive Normal Form (CNF), defined over a set X of n Boolean variables $\{x_1, \dots, x_n\}$, is a conjunction of clauses, where a clause is a disjunction of literals. A literal is either a variable x_i or its complement $\neg x_i$. The Propositional Satisfiability (SAT) problem consists of deciding whether there exists an assignment to the variables in X such that the formula ϕ is satisfied. For ease of explanation, we often use the set notation to represent the conjunction of clauses and formulas.

As an example, consider the CNF formula $\phi = \{(\neg x_1 \vee x_2), (x_1 \vee x_2 \vee x_3), (\neg x_2 \vee \neg x_3)\}$. In this case, a possible satisfying assignment would be $x_1 = 0, x_2 = 0, x_3 = 1$, since it satisfies all clauses in ϕ.

The Satisfiability Modulo Theories (SMT) problem is a generalization of SAT. Given a decidable first order theory $\mathcal{T}$, a $\mathcal{T}$-atom is a ground atomic formula in $\mathcal{T}$. A $\mathcal{T}$-literal is either a $\mathcal{T}$-atom t or its complement $\neg t$. A $\mathcal{T}$-formula is similar to a propositional formula, but a $\mathcal{T}$-formula is composed of $\mathcal{T}$-literals instead of propositional literals. Given a $\mathcal{T}$-formula ϕ, the SMT problem consists of deciding if there exists a total assignment over the variables of ϕ such that ϕ is satisfied. Depending on the theory $\mathcal{T}$, the variables can be of type integer, real, Boolean, etc.

Using SMT to encode multi-threaded executions. Some debugging systems represent multi-threaded executions as SMT formulations over symbolic variables (Huang, Zhang,

[1]We refer to the literature for a more detailed background on concurrency bugs (Lu et al. 2006; Park, Vuduc, and Harrold 2010).

```
                 (initially x = y = w = z = 0)
        T1                 T2              T3
1. if (z > 0)        1. z = 1         1. if (w > 0)
2.    w++            2. x = x + 1     2.    y = 0
3. x = 1             3. y = y + 1
4. y = 1
5. if (y == 0)
6.    x = x - 1
7. assert(x > 0)
```

Figure 2: A multi-threaded program with a path and schedule-dependent bug.

```
                 (initially x = y = w = z = 0)
        T1                 T2              T3
                     1. z = 1
                     2. x = x + 1
                     3. y = y + 1
1. [z > 0]
2.    w++
3. x = 1
4. y = 1
                                      1. [w > 0]
                                      2.    y = 0
5. [y == 0]
6.    x = x - 1
7. assert(x > 0)
```

Figure 3: Failing schedule for program in Figure 2.

and Dolby 2013; Machado, Lucia, and Rodrigues 2015; 2016). The constraints of a concurrent execution are composed by a set of sub-formulae that encode, respectively, the path conditions, the program order of operations in each individual thread, the read-write linkages of shared variables, the synchronization points, and the failure condition.

Let $var_{t,l}$ denote the value of a shared variable var at line l of thread t. Consider the initial values of the variables as having line and thread equal to 0. Moreover, let the *order variables* $R_{t,l}$ ($W_{t,l}$) denote the execution order of the read (write) operation at line l of thread t. A concrete instance of the SMT formula for the multi-threaded program in Figure 2 is as follows.[2]

Path Constraints. Let us assume an execution path in which the branch condition at line 1 of thread T1 evaluates to $true$, whereas the branch conditions at lines 5 and 1 of threads T1 and T3, respectively, evaluate to $false$. The path constraints are then encoded as:

$$(z_{1,1} > 0) \wedge \neg(y_{1,5} = 0) \wedge \neg(w_{3,1} > 0)$$

Program Order Constraints. The following SMT formula enforces the schedule of operations in a particular thread:

$$(R_{1,1} < W_{1,2} < W_{1,3} < W_{1,4} < R_{1,5} < R_{1,7}) \wedge$$
$$(W_{2,1} < W_{2,2} < W_{2,3}) \ \wedge (R_{3,1} < W_{3,2})$$

[2]Since the program in Figure 2 does not have synchronization points, we do not consider this type of constraints in the model description for the example.

Read-Write Constraints. To encode all possible data-flows over shared variables due to reads and writes performed by the three threads along their execution paths, one would add the following conjunction of constraints to the system:

$$\underbrace{(x_{0,0} = 0) \wedge (y_{0,0} = 0) \wedge (w_{0,0} = 0) \wedge (z_{0,0} = 0)}_{\text{initial values}}$$

$$\underbrace{\begin{gathered}((z_{1,1} = z_{0,0} \wedge R_{1,1} < W_{2,1}) \vee \\ (z_{1,1} = z_{2,1} \wedge R_{1,1} > W_{2,1})) \wedge (z_{2,1} = 1)\end{gathered}}_{\text{reads/writes on } z}$$

$$\underbrace{\begin{gathered}(w_{1,2} = w_{0,0} + 1) \wedge \\ ((w_{3,1} = w_{0,0} \wedge R_{3,1} < W_{1,2}) \vee (w_{3,1} = w_{1,2} \wedge R_{3,1} > W_{1,2}))\end{gathered}}_{\text{reads/writes on } w}$$

$$\underbrace{\begin{gathered}(y_{1,4} = 1) \wedge \\ ((y_{1,5} = y_{1,4} \wedge (R_{1,5} < W_{2,3} \vee W_{1,4} > W_{2,3})) \vee \\ (y_{1,5} = y_{2,3} \wedge R_{1,5} > W_{2,3} \wedge W_{2,3} > W_{1,4})) \wedge \\ ((y_{2,3} = y_{0,0} + 1 \wedge R_{2,3} < W_{1,4}) \vee (y_{2,3} = y_{1,4} + 1 \wedge R_{2,3} > W_{1,4}))\end{gathered}}_{\text{reads/writes on } y}$$

$$\underbrace{\begin{gathered}(x_{1,3} = 1) \wedge \\ ((x_{1,7} = x_{1,3} \wedge (R_{1,7} < W_{2,2} \vee W_{1,3} > W_{2,2})) \vee \\ (x_{1,7} = x_{2,2} \wedge R_{1,7} > W_{2,2} \wedge W_{2,2} > W_{1,3})) \wedge \\ ((x_{2,2} = x_{0,0} + 1 \wedge R_{2,2} < W_{1,3}) \vee (x_{2,2} = x_{1,3} + 1 \wedge R_{2,2} > W_{1,3}))\end{gathered}}_{\text{reads/writes on } x}$$

Note that each conjunct expresses the possible values returned by a read operation on a shared variable by encoding the most-recent write to that variable. For instance, the constraints $(x_{2,2} = x_{0,0} + 1 \ \wedge \ R_{2,2} < W_{1,3}) \ \vee (x_{2,2} = x_{1,3} + 1 \wedge R_{2,2} > W_{1,3})$ indicate that the read on x by T2 at line 2 either returns the result of adding 1 to the initial value (therefore, read $R_{2,2}$ happens before write $W_{1,3}$) or the result of T1's write at line 3 (hence, write $W_{1,3}$ happens before read $R_{2,2}$ and is the most recent write to x).

Failure Constraint. Finally, to force the solver to produce a failing schedule, one adds a constraint encoding the *negation* of the assertion: $\neg(x_{1,7} > 0)$.

In contrast, to obtain a non-failing schedule, one would add the original assertion to the formulation instead of the constraint above. Furthermore, one would also need to add constraints limiting the order variables to have all distinct integer values within the range $[1, n]$, where n is the number of operations of the program. In this example, the Linear Integer Arithmetic (LIA) theory is being used as $\mathcal{T}$.

Using SMT in related areas. Over the last decade, there has been a growing interest in using SMT for concurrency debugging. For instance, several record and replay systems use SMT to deterministically reproduce concurrency bugs (Lee et al. 2009; Altekar and Stoica 2009; Huang, Zhang, and Dolby 2013). Alongside, BugAssist (Jose and Majumdar 2011) pioneered root cause isolation with SMT constraint solving by leveraging unsatisfiable cores. However, BugAssist is only able to diagnose errors in sequential programs. ConcBugAssist (Khoshnood, Kusano, and Wang 2015) extends BugAssist to handle concurrency bugs and automatically generate fixes by casting the binate covering problem as a constraint formulation. Finally, SMT has also been extensively used in testing (Huang 2015; Farzan et al. 2013) of concurrent programs.

2.3 Differential Schedule Projections

In the context of root cause isolation using SMT solving, the SYMBIOSIS debugging tool (Machado, Lucia, and Rodrigues 2015) introduced the notion of Differential Schedule Projection (DSP). Like the name suggests, a DSP is built by projecting a failing schedule onto an alternate schedule (which is a non-failing variant of the failing schedule). The DSP prunes out the common operations in the projection and reports solely the subset of thread events that differ between the schedules. These events correspond to read-write linkages that cause the failure in the failing schedule.

More formally, let us define an execution schedule as a 3-tuple (V, E_s, E_{df}), corresponding to a directed acyclic graph (DAG) with a set V of vertices, representing thread operations in the execution, and two sets of directed edges connecting the vertices: *schedule edges* (E_s) and *data-flow edges* (E_{df}). A schedule edge links a pair of operations, belonging to the same thread or not, according to the execution order given by the schedule. In turn, a data-flow edge connects a write operation w to a read operation r, thus indicating that r reads the value written by w during the execution.

Consider now that $F = (V, E_s^F, E_{df}^F)$ and $A = (V, E_s^A, E_{df}^A)$ denote the DAGs of the failing and alternate schedules, respectively. The portions of the schedules relevant to the bug's root cause are obtained by computing the union of the edges exclusive to both schedules: $E_s^p = (E_s^F \cup E_s^A) \setminus (E_s^F \cap E_s^A)$ and $E_{df}^p = (E_{df}^F \cup E_{df}^A) \setminus (E_{df}^F \cap E_{df}^A)$. The DSP is then given by the pair (F^p, A^p), where $F^p = (V^p, E_s^F \cap E_s^p, E_{df}^F \cap E_{df}^p)$, $A^p = (V^p, E_s^A \cap E_s^p, E_{df}^A \cap E_{df}^p)$, and V^p is the set of vertices (i.e. operations) appearing in the edges in both E_s^p and E_{df}^p.

By highlighting the data-flow variations between the schedules, the DSP allows obviating most of the execution's complexity (which is often irrelevant for the failure to occur) and steer the programmer's focus to the bug's root cause.

2.4 Differential Path-Schedule Projections

When the manifestation of the failure depends on the threads' execution path, an alternate, non-failing schedule will necessarily exhibit a variation in the program's control-flow with respect to the failing schedule. To handle such path-schedule dependent bugs, CORTEX (Machado, Lucia, and Rodrigues 2016) introduced Differential Path-Schedule Projections (DPSPs), which are an extension of DSPs that highlight differences in the control-flow, in addition to differences in the data-flow.

However, since a change in the control-flow of a multi-threaded program is, in practice, the result of a variation in the schedule of the thread operations that changes the data-flow on shared variables and causes a given branch condition to evaluate differently, in the rest of the paper we will use the term DSP to refer to both a DSP and a DPSP.

It should also be noted that the accuracy of DSPs is heavily determined by the similarity between the schedules. The more different the thread interleaving of the failing schedule is from that of the alternate schedule, the fewer common portions their projection will exhibit and the longer the resulting DSP will be. An unnecessarily long DSP will contain events from the original execution that are not related to the bug's root cause and, consequently, hinder the debugging task. It is thus of paramount importance to obtain failing and alternate schedules as identical as possible, in order to generate an accurate and useful differential projection.

3 SYMBIOSIS and CORTEX Shortcomings

This section describes systems SYMBIOSIS' and CORTEX's approaches for generating Differential Schedule Projections (DSPs), and highlights their key shortcomings.

3.1 Swapping One Operation Pair is not Enough

SYMBIOSIS leverages symbolic execution and SMT constraint solving to produce DSPs that isolate concurrency bugs. SYMBIOSIS receives as input a set of lightweight per-thread path profiles recorded from an original failing execution. Next, it performs a guided symbolic execution along the thread path profiles with the goal of generating per-thread traces with symbolic information. The symbolic traces are used to generate an SMT constraint formulation encoding the failure condition, as described in Section 2.2. Afterwards, SYMBIOSIS resorts to an SMT solver to solve the constraint system and output a failure-inducing ordering of operations, i.e. the failing schedule.

After obtaining the failing schedule, SYMBIOSIS has to devise an alternate, non-failing schedule in order to be able to compute the DSP and pinpoint the bug's root cause. The generation of the alternate schedule is done by applying the following heuristic:

1. Select a pair of events in the failing schedule, prioritizing pairs whose events are closer in the execution;
2. Generate a candidate alternate schedule by swapping the order of the two events in the failing schedule;
3. Check the satisfiability of the candidate alternate schedule by solving an SMT formula like the one used to produce the failing schedule, but modified to have the failure constraint inverted and new constraints enforcing the order of operations given by the candidate alternate schedule;
4. If the SMT formula is satisfiable, return the non-failing schedule. Otherwise, repeat steps 1 to 3 until a feasible alternate schedule is found, or until all possible event pairs have been reordered (in which case, SYMBIOSIS fails).

Once a feasible alternate schedule is found, SYMBIOSIS proceeds to computing the DSP as described in Section 2.3. Experimental results showed that this heuristic works well in various scenarios (Machado, Lucia, and Rodrigues 2015). However, it falls short for cases where the failure may stem from more than one data-flow.

As an example of this limitation, recall the buggy program in Figure 1. Note that it suffices that T4 runs before T2 or T5 runs before T3 for the program to fail. Consider the failing schedule of operations <T4, T5, T2, T3> for Figure 1. SYMBIOSIS will attempt to generate an alternate schedule by reordering the following pairs of instructions:

(initially $x = y = w = z = 0$)

T1	T2	T3
		1. $[\neg(w > 0)]$
	1. $z = 1$	
1. $[z > 0]$		
2. $w++$		
3. $x = 1$		
	2. $x = x + 1$	
4. $y = 1$		
	3. $y = y + 1$	
5. $[\neg(y == 0)]$		
7. assert($x > 0$)		

Figure 4: Possible alternate schedule for Figure 3.

(initially $x = y = w = z = 0$)

T1	T2	T3
	1. $z = 1$	
	2. $x = x + 1$	
	3. $y = y + 1$	
1. $[z > 0]$		
		1. $[\neg(w > 0)]$
2. $w++$		
3. $x = 1$		
4. $y = 1$		
5. $[\neg(y == 0)]$		
7. assert($x > 0$)		

Figure 5: An alternate schedule for Figure 3 that results in a smaller DPSP than the one in Figure 4.

$(R_{1,5}, W_{3,1})$, $(R_{1,5}, W_{2,1})$, $(R_{3,1}, W_{2,1})$, $(R_{3,1}, W_{5,1})$, and $(R_{2,1}, W_{4,1})$. Swapping the former two event pairs renders the model unsatisfiable because it violates the synchronization semantics of the program, which state that the end of a thread must happen before the $join$ operation by the parent thread. In turn, inverting the order of the remaining pairs will fix one of the two problematic data-flows ($z = z + 1$ before $x = x + z$ or $y = y + 1$ before $x = x + y$), but never both simultaneously. For that reason, SYMBIOSIS will fail to find a feasible alternate schedule.

3.2 The Disregard for Minimality

Conversely to SYMBIOSIS, CORTEX aims at exposing and isolating concurrency bugs in *correct* executions. CORTEX starts from a set of per-thread path profiles collected from one or more executions and produces a non-failing schedule using symbolic execution and SMT constraint solving. In fact, the model built by CORTEX is similar to the one employed by SYMBIOSIS to obtain the initial failing schedule.

Afterwards, CORTEX systematically explores the state space of the program in an attempt to unveil a schedule that leads to a failure. More concretely, CORTEX inverts branch conditions in the original non-failing schedule, starting from branches that are closer to the assertion point. For each new combination of thread symbolic traces, CORTEX builds an SMT formula encoding the failure condition and checks its satisfiability. If the formula is satisfiable, then CORTEX obtains the failing schedule and proceeds to the computation of the DSP. Otherwise, CORTEX flips a different branch condition, synthesizes new thread traces via symbolic execution, and repeats the procedure until a maximum number of branch flips is reached.

Using this approach, CORTEX is able to generate failing schedules for failures that are path-schedule dependent. Then, an alternate schedule is produced through an SMT solver, as in SYMBIOSIS. Branch conditions are flipped and new thread traces synthesized if necessary. Unfortunately, the alternate schedule found by the SMT solver does not come with any guarantees in terms of similarity to the initial non-failing schedule. In fact, CORTEX simply uses the first alternate schedule produced by the solver.

To better illustrate this limitation, recall the program in Figure 2. Figure 3 shows a possible failing schedule for this program. Recall that the failure occurs because T3 sets y to 0 before the evaluation of the branch condition in line 5 of T1, which causes x to be decremented to 0 at line 6 and, consequently, the violation of the assertion. For the purpose of this example, let us consider that CORTEX starts from the failing schedule in Figure 3 and aims at finding an alternate schedule that allows computing the DSP. Since this failure is path-schedule dependent, CORTEX needs to synthesize new execution traces in order to find a feasible alternate schedule.

Suppose that CORTEX synthesizes an execution that flips the path conditions of lines 1 and 5 of T3 and T1, respectively. The alternate schedules depicted in Figures 4 and 5 are two possible interleavings found by the SMT solver for this scenario. The DSPs generated for these two alternate schedules would differ. The alternate schedule in Figure 5 is more similar to the failing schedule, as they share a longer common prefix (namely all the instructions executed by T2). Since the common prefix is irrelevant to the occurrence of the bug, it can be discarded when computing the DSP.

4 Generating DSPs with MaxSMT

This section proposes an approach based on Maximum Satisfiability to address the limitations in SYMBIOSIS and CORTEX presented in Section 3.

4.1 Maximum Satisfiability in SMT

The *Maximum Satisfiability* (MaxSAT) problem can be seen as an optimization version of the SAT problem. In MaxSAT, the goal is to find an assignment to the variables of a CNF formula that maximizes the number of satisfied clauses. There are several variants of MaxSAT (Li and Manyà 2009). In the context of this work, we consider the *partial MaxSAT problem* where a MaxSAT formula $\phi = \phi_h \cup \phi_s$ has some clauses considered as *hard* (ϕ_h), while others are declared as *soft* (ϕ_s). In partial MaxSAT, the goal is to find an assignment to the variables in ϕ such that all hard clauses are satisfied and the number of satisfied soft clauses is maximized.

One can also define the Maximum Satisfiability problem in SMT formulas. The *MaxSMT* problem is similar to MaxSAT, except that ϕ_h, as well as ϕ_s contain $\mathcal{T}$-formulas instead of propositional clauses. In MaxSMT, the goal is to find an assignment that satisfies all $\mathcal{T}$-formulas in ϕ_h and maximizes the number of satisfied soft $\mathcal{T}$-formulas in ϕ_s.

Example 4.1. *Consider the MaxSMT formula $\phi = \phi_h \cup \phi_s$ where $\phi_h = \{(x_1 \geq 0), (x_1 \leq 4)\}$ and $\phi_s = \{(x_2 \geq -1), (x_2 \leq 0), (x_1 + x_2 = 5)\}$. Here, $\mathcal{T}$ is the Linear Integer Arithmetic (LIA) theory. An optimal solution would be $x_1 = 4, x_2 = 1$, where soft constraint $(x_2 \leq 0)$ is not satisfied.*

Observe that algorithms to solve MaxSAT that use iterative calls to a SAT solver can be adapted to solve MaxSMT. In MaxSMT, an SMT solver is used instead of a SAT solver. For instance, the SMT solver Z3 (de Moura and Bjørner 2008) is able to solve MaxSMT formulas (Bjørner and Narodytska 2015). In the last decade, SMT solvers became able to solve much larger problem instances than previously. As a result, concurrency debuggers such as CLAP (Huang, Zhang, and Dolby 2013), SYMBIOSIS and CORTEX have successfully used SMT constraint solving to achieve their goals. However, these tools rely on heuristic procedures with no guarantee of approximation to the optimum.

One should note that soft formulas encode objectives or preferences. It is well-known that any linear objective function can be easily encoded by using a set of soft formulas in MaxSAT or MaxSMT. However, there exists a vast number of applications where there is more than one objective function to optimize. When there are multiple objectives, one can define a different set of soft formulas for each individual objective. Moreover, if those objectives can be sorted according to some criteria, then each objective (encoded as a set of soft formulas) can be optimized in order by using lexicographic optimization (Marques-Silva et al. 2011).

4.2 Finding an Optimal DSP

In this section, we propose using MaxSMT to compute alternate schedules as similar as possible to the corresponding failing schedules. Those alternate schedules can then be used to generate simpler and more informative DSPs than the ones currently produced by the SYMBIOSIS and CORTEX systems. We consider three *DSP quality* criteria when measuring schedule similarity:

- *Number of data-flow variations*. This corresponds to the amount of data-flows that appear in the alternate schedule but not in the failing schedule, i.e. $\left|E^A_{df} \cap E^p_{df}\right|$.
- *Number of broken segments*. Let a segment $S = o_1 o_2 \ldots o_k$ be a maximal sequence of operations in the failing schedule such that all o_i belong to the same thread T and for all o' not belonging to T we have that o' occurs before o_1 or after o_k. S is considered *broken* if, in the alternate schedule, some operation of a thread $T' \neq T$ appears after o_1 but before o_k.
- *Number of context switch variations*. This accounts for the context switches in the failing schedule that do not occur in the alternate schedule, i.e., contiguous pairs of operations $o_i o_j$ in the failing schedule such that o_i and o_j belong to different threads and, in the alternate schedule, some other operation o_k appears in-between o_i and o_j.

The rationale for the aforementioned DSP quality criteria stems from the observation that most real-world concurrency bugs can be triggered by only a few thread context switches (Musuvathi et al. 2008). By generating an alternate schedule that minimizes the variation in the data-flow and context switching with respect to the failing sequence of operations, one is able to reduce the cardinality of the projection edge sets E^p_{df} and E^p_s and thus produce a DSP that accurately isolates the bug's root cause.

The soft formulas in our MaxSMT model encode the minimization of these criteria, while constraints presented in section 2.2 are considered as hard formulas. For each data-flow edge $(w, r) \in E^F_{df}$, we add a soft formula stating that operation r must read the value written by w. Given some operation o, we denote its order variable as O. Therefore, let R and W be the order variables of r and w respectively. For every data-flow edge (w, r), the MaxSMT model considers, as a soft formula, the conjunction of $W < R$ with $W' < W \vee W' > R$ for all $w' \neq w$ such that w' is a write operation on the variable read by r. To encode the broken segments criteria, we consider every possible maximal sequence $S = o_1 o_2 \ldots o_k$ of operations, in the failing schedule, belonging to the same thread, and add the soft formula $O_k - O_1 = k - 1$. Finally, in order to minimize the number of context switch changes, for each schedule edge (o_i, o_j) such that o_i and o_j belong to different threads, the model considers the soft formula $O_i = O_j - 1$.

We can look at this schedule computation problem as a multi-objective problem with three different sets of soft formulas ϕ^{df}_s, ϕ^{bs}_s and ϕ^{cs}_s, representing the data-flows, broken segments and context switches criteria respectively. We consider two types of MaxSMT models: one where all criteria are equally significant (i.e., $\phi_s = \phi^{df}_s \cup \phi^{bs}_s \cup \phi^{cs}_s$) and one lexicographic model that optimizes ϕ^{df}_s first, followed by ϕ^{bs}_s and then ϕ^{cs}_s. By prioritizing the optimization of the data-flows, one ensures that the specific interleaving of operations that causes the bug is the most accurate possible. Having that, the optimization of the other criteria is mainly to allow pruning out additional thread segments that, although differing in the two executions, are not relevant to the failure.

Differential Path-Schedule Projections. Recall from section 2.4 that, if the failure is path dependent, CORTEX synthesizes execution traces that follow a different control-flow than the one in the original non-failing schedule. As a result, some operations in the original schedule may not appear in new traces and vice-versa. When building ϕ^{df}_s, we ignore data-flow edges containing at least one operation that does not appear in the synthesized traces. Same goes for operation sequences (pairs) considered when building ϕ^{bs}_s (ϕ^{cs}_s).

4.3 Approximating an Optimal DSP

In our experiments, we observed that the MaxSMT approach proposed in the previous section struggled to find alternate schedules within reasonable time. However, in practice, one does not necessarily have to find the alternate schedule that is the most similar to the failing schedule in order to produce a DSP that is sufficiently short and accurate.

State-of-the-art MaxSMT algorithms rely on multiple calls to an SMT oracle and compute intermediate sub-

Algorithm 1: Progressive algorithm for computing an alternate schedule.

```
Input: $F$
1  $S_a \leftarrow$ AssertSegment($F$)
2  $OP_{fix} \leftarrow$ Operations($F$) $\setminus S_a$
3  $S_f \leftarrow S_a; S_l \leftarrow S_a$
4  $A \leftarrow nil$
5  do
6      $S_f \leftarrow$ PreviousSegment($F$, $S_f$)
7      $S_l \leftarrow$ NextSegment($F$, $S_l$)
8      $OP_{fix} \leftarrow OP_{fix} \setminus (S_f \cup S_l)$
9      $A \leftarrow$ FindAltSchedule($F$, $OP_{fix}$)
10 while $A = nil \wedge OP_{fix} \neq \emptyset$;
11 return $A$
```

optimal solutions during the search process (Bjørner and Phan 2014). Internally, modern SMT solvers apply an adaptation of the conflict-driven DPLL procedure used in state-of-the-art SAT solvers (Sebastiani 2007). A simple approach is to terminate the search early if the number of conflicts crosses a fixed threshold. In such case, the MaxSMT solver returns the best solution it was able to find. If no solution was found, the formula is assumed to be unsatisfiable.

A better approach is to generate simpler MaxSMT models by first focusing on the operations close to the assertion and progressively expanding until an alternate schedule is found. This is the idea behind Algorithm 1, which receives a failing schedule F as input and outputs an alternate schedule A. Recall that a failing schedule is a sequence of operations that results in a failure and segments are maximal sequences of operations belonging to the same thread. The assertion segment S_a is the segment that includes the failing assertion. Algorithm 1 starts by setting OP_{fix} to all operations except those in S_a (line 1). At each iteration, the order of the operations in OP_{fix} is fixed. The algorithm then retrieves the segments immediately before (S_f) and after (S_l) the assertion segment (lines 6 and 7). Next, it unfixes the corresponding operations from OP_{fix} (line 8) and attempts to compute an alternate schedule by solving a MaxSMT model (line 9). If an alternate schedule is found, then that schedule is returned. Otherwise, the algorithm keeps removing operations from OP_{fix} until it either finds an alternate schedule or the set of fixed operations becomes empty (line 10). Note that conflict thresholds can also be used in algorithm 1 in the computation of the alternate schedule in order to avoid getting stuck solving hard MaxSMT formulas.

5 Experimental Evaluation

In order to evaluate the proposed MaxSMT-based approaches, the MaxSMT models and algorithms were integrated in the publicly available versions of SYMBIOSIS[3] and CORTEX[4] tools. The experimental evaluation is mainly focused on assessing the *performance* and the *quality of the DSPs* produced by our MaxSMT-based approaches with respect to the base versions of SYMBIOSIS and CORTEX. The performance criterion is measured in terms of constraint solving time. For the latter criterion, we assume that reducing the size of the DSPs and the values of the metrics discussed in section 4.2 improves the quality of the DSP and the productivity of the programmer when fixing the bug.

For the comparison, we resorted to several buggy multi-threaded programs commonly used in the literature. We consider two sets of test cases. The first includes several C/C++ and Java applications with strictly schedule-dependent bugs, such as shared buffer implementation or an adapted parallel file scanner (Elmas et al. 2013). The second set includes programs with path-schedule dependent bugs from the IBM ConTest benchmark suite, as well as the *ExMCR* benchmark.

In our experiments, Z3 (Bjørner and Phan 2014) (version 4.6.0) was used as the MaxSMT solver. Z3 has native support for lexicographic optimization and implements MaxSMT algorithms MaxRes and WMax.

5.1 Performance Comparison

Table 1 shows the execution times of the heuristics of SYMBIOSIS/CORTEX and the MaxSMT-based approaches. Columns 'MaxSMT' and 'Lexico' correspond to the full MaxSMT and lexicographic models respectively. The 'Prog (X)' columns correspond to the progressive algorithm (Algorithm 1), where X stands for the type of MaxSMT model used. The 'cmp' (complete) sub-column indicates that no conflict threshold was imposed. For the 'inc' (incomplete) sub-column, a conflict threshold of 200000 was used. Each line corresponds to a different test case and the best entries for each test case are highlighted. A time limit of 3 hours was imposed for each execution on a particular test case. Entries with 'TO' indicate that the algorithm timed out, while entries with 'FAIL' indicate that the algorithm terminated within the time limit, but failed to find an alternate schedule.

The results in Table 1 show that algorithms using the full MaxSMT model require, in general, more time to generate the DSPs. However, the progressive algorithm is always as fast or much faster than all remaining approaches, including the heuristic of SYMBIOSIS/CORTEX. In particular, for the *bufwriter* test case, all configurations of the progressive algorithm generate the DPSP at least 60 times faster than the heuristic. The only exception is the *manager* test case, but our approach is able to produce a much improved DPSP (see Table 2). The 'Average' line shows the average execution time for each algorithm only considering solved instances. However, some algorithms are not able to solve all instances within the time limit. These entries are underlined. Nevertheless, the average execution time of the progressive algorithm is much smaller than the remaining approaches.

5.2 DSP Quality Comparison

Table 2 shows the quality of the DSPs found by the original version of SYMBIOSIS/CORTEX and by the proposed MaxSMT-based approaches. We consider the number of operations, data-flow variations, broken segments and context switch variations in the DSP. Results are shown only for a representative set of configurations (see Table 3) which exhibit better performance. The 'Average' line shows the average relative DSP quality improvement/degradation with re-

[3]http://github.com/nunomachado/symbiosis

[4]http://github.com/nunomachado/cortex-tool

Table 1: Time in seconds required to generate DSPs (DPSPs) using SYMBIOSIS/CORTEX and MaxSMT

Test Case	SYMB/	MaxSMT				Lexico				Prog (MaxSMT)				Prog (Lexico)			
		MaxRes		WMax		MaxRes		WMax		MaxRes		WMax		MaxRes		WMax	
	CORTEX	cmp	inc	cmp	inc	cmp	inc	cmp	inc	cmp	inc	cmp	inc	cmp	inc	cmp	inc
crasher	22.6	3373	FAIL	4275	FAIL	4321	FAIL	TO	FAIL	**0.4**	**0.4**	**0.4**	**0.5**	**0.4**	**0.4**	**0.4**	**0.4**
pbzip2 (S)	**0.3**	22.8	22.8	19.9	20.1	37.5	38.3	38.2	38.0	**0.2**	**0.2**	**0.2**	**0.2**	**0.2**	**0.2**	**0.2**	**0.2**
pbzip2 (M)	**0.9**	122.8	122	112	112	TO	1280	4306	3751	**0.4**	**0.4**	**0.4**	**0.4**	**0.3**	**0.3**	**0.3**	**0.3**
pbzip2 (L)	28.7	TO	TO	TO	TO	TO	TO	TO	TO	**2.3**	**2.3**	**2.3**	**3.0**	**2.2**	**2.2**	**2.2**	**2.1**
pfscan	**0.2**	812	99.1	2.1	2	**0.2**	**0.6**	**0.4**	**0.7**	**0.1**	**0.1**	**0.1**	**0.2**	**0.1**	**0.1**	**0.1**	**0.1**
airline	**0.2**	TO	88.9	2.1	1.7	11.7	11.7	1.0	1.0	**0.1**	**0.1**	**0.1**	**0.2**	**0.1**	**0.1**	**0.1**	**0.1**
bank	FAIL	11.6	11.8	3.3	3.4	2.5	2.6	2.2	2.2	**1.5**	**1.4**	**1.1**	**1.1**	**1.2**	**1.2**	**1.1**	**1.1**
cache4j (M)	114.1	3048	3301	3508	3794	3489	3494	2821	2846	**6.0**	**5.9**	6.7	7.8	**6.0**	**5.3**	6.9	7.0
cache4j (L)	343	TO	TO	TO	10668	TO	TO	TO	TO	176	166	19.9	20.4	**16.2**	**16.6**	109	99.6
bufwriter	5790	TO	TO	8939	8940	3773	3828	5803	5782	91.7	91.5	**32.9**	**33.0**	60.6	62.6	38.5	38.4
critical	**0.5**	2.5	**0.3**	2.3	**0.2**	**0.1**	**0.3**	**0.2**	**0.3**	**0.2**	**0.7**	2.6	**0.2**	**0.2**	**0.2**	**0.3**	**0.6**
exMCR	**1.1**	**0.6**	**0.6**	**0.2**	**0.2**	**0.2**	**0.6**	2.1	2.2	**0.5**	**0.7**	**0.6**	**0.6**	**0.4**	**0.6**	**0.6**	**0.6**
garage	13.4	TO	673	39.7	40.8	62.9	59.6	77.3	76.8	**0.5**	**0.4**	**0.5**	**0.4**	**0.4**	**0.4**	**0.5**	**0.4**
loader	55.8	107	102	239	246	130	135	TO	701	97.9	96.8	**10.5**	**10.5**	22.4	22.0	31.3	31.2
manager	**189**	TO	FAIL	TO	FAIL	TO	FAIL	TO	FAIL	TO	2171	TO	1630	TO	1567	TO	1199
ticketorder	7.1	1101	1097	373	371	182	186	146	146	9.1	9.1	**3.7**	**3.6**	12.7	13.0	**3.3**	**3.4**
Average	437	782	502	1347	1862	1001	753	1200	1112	25.8	159	5.5	107	8.2	106	13	**86.5**

Table 2: Quality of the DSPs (DPSPs), generated by SYMBIOSIS/CORTEX and MaxSMT, measured using the number of operations (OP), data-flow variations (DF), broken segments (BS) and context switch variations (CTX)

Test Case	SYMB / CORTEX				MaxSMT				Lexico				Prog (MaxSMT)				Prog (Lexico)			
	OP	DF	BS	CTX	OP	DF	BS	CTX	OP	DF	BS	CTX	OP	DF	BS	CTX	OP	DF	BS	CTX
crasher	**9**	**1**	1	**2**	**9**	**1**	1	**2**	**9**	**1**	**0**	3	**9**	**1**	1	**2**	**9**	**1**	**0**	3
pbzip2 (S)	**7**	**1**	2	**1**	**7**	**1**	**1**	**1**	**7**	**1**	**1**	**1**	**7**	**1**	**1**	**1**	**7**	**1**	**1**	**1**
pbzip2 (M)	4	**1**	2	**1**	**3**	**1**	1	**1**	TO				**3**	**1**	**0**	2	**3**	**1**	**0**	2
pbzip2 (L)	**7**	**1**	2	**1**	TO				TO				**7**	**1**	**1**	**1**	**7**	**1**	**1**	**1**
pfscan	**27**	**1**	2	**1**	**27**	**1**	1	**1**	**27**	**1**	**0**	2	**27**	**1**	**0**	2	**27**	**1**	**0**	2
airline	11	2	2	**1**	**3**	**1**	**1**	**1**	**3**	**1**	**1**	**1**	8	2	**1**	**1**	8	2	**1**	**1**
bank	FAIL				**6**	**1**	**1**	2	**6**	**1**	**1**	2	54	2	2	**1**	54	2	**1**	2
cache4j (M)	14	**1**	1	**2**	**3**	**1**	1	**2**	**3**	**1**	**0**	3	43	**1**	1	**2**	**3**	**1**	**0**	3
cache4j (L)	5	**1**	2	**1**	TO				TO				**3**	**1**	**1**	2	**3**	**1**	**1**	2
bufwriter	724	88	7	6	197	**1**	2	**1**	**35**	**1**	**1**	2	221	8	3	3	221	8	3	3
critical	9	**4**	3	4	**7**	**4**	**2**	**2**	**7**	**4**	**2**	**2**	**7**	**4**	**2**	**2**	**7**	**4**	**2**	**2**
exMCR	22	8	8	12	**21**	**7**	6	**8**	**21**	**7**	**5**	9	**21**	**7**	6	**8**	**21**	**7**	**5**	9
garage	5	2	1	2	**3**	**1**	2	**1**	**3**	**1**	1	2	5	2	**0**	3	5	2	**0**	3
loader	267	22	18	18	48	**21**	3	**2**	**47**	**21**	**2**	3	48	**21**	4	**2**	**47**	**21**	3	3
manager	120	46	58	147	TO				TO				78	33	34	109	**57**	**32**	**9**	**70**
ticketorder	178	**4**	5	5	**46**	**4**	**1**	2	**46**	**4**	**1**	2	47	**4**	2	**1**	**46**	**4**	**1**	2
Average (%)	-	-	-	-	61	82	67	**70**	**57**	**80**	34	92	89	91	47	102	68	91	**28**	109

spect to SYMBIOSIS/CORTEX. For example, 61% in the OPs column for 'MaxSMT' indicates that the alternate schedule found produces, on average, a DSP with 100% - 61% = 39% fewer operations than that of SYMBIOSIS/CORTEX.

In general, using the full MaxSMT model results in higher quality DSPs. For example, when 'Lexico' is able to find an alternate schedule, it produces a DSP with the smallest number of operations, data-flow variations and broken segments. 'MaxSMT' performs better in terms of context switches, which is expected since 'Lexico' prioritizes data-flow variations and broken segments. The DSPs produced by the progressive approaches are, usually, of equal quality to those produced with the full models. Moreover, all the MaxSMT-based approaches outperform SYMBIOSIS/CORTEX, especially in the test cases with path-schedule dependent bugs.

6 Conclusions

This paper proposes the use of Maximum Satisfiability Modulo Theories (MaxSMT) to generate minimal Differential Schedule Projections (DSPs) that automatically provide an explanation of a concurrency bug in a multi-threaded program. Also, in cases where solving the MaxSMT model is

Table 3: Representative configurations per type of approach

Approach	MaxSMT Algorithm	Conflict Threshold
MaxSMT	WMax	No
Lexico	MaxRes	No
Prog (MaxSMT)	WMax	Yes
Prog (Lexico)	WMax	Yes

too time-consuming, we exploit domain knowledge to build smaller models and reduce the constraint solving time while still providing a close approximation to an optimal DSP. perations The proposed MaxSMT-based approaches were integrated into two state-of-the-art systems SYMBIOSIS and CORTEX. Experimental results, using benchmark and real-world concurrency bugs, show that using a MaxSMT model greatly improves the quality of the DSPs, especially when bugs are path and schedule dependent.

Acknowledgments

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UID/CEC/50021/2013, SFRH/BD/111471/2015, CMU/AIR/0022/2017 and PTDC/CCI-COM/31198/2017.

References

Altekar, G., and Stoica, I. 2009. ODR: output-deterministic replay for multicore debugging. In *22nd Symposium on Operating Systems Principles*, 193–206. ACM.

Berger, E. D.; Yang, T.; Liu, T.; and Novark, G. 2009. Grace: safe multithreaded programming for C/C++. In *24th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 81–96. ACM.

Bjørner, N., and Narodytska, N. 2015. Maximum satisfiability using cores and correction sets. In *International Joint Conference on Artificial Intelligence*, 246–252. AAAI Press.

Bjørner, N., and Phan, A. 2014. νZ - maximal satisfaction with Z3. In *6th International Symposium on Symbolic Computation in Software Science*, 1–9.

de Moura, L. M., and Bjørner, N. 2008. Z3: an efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 337–340. Springer.

Devietti, J.; Lucia, B.; Ceze, L.; and Oskin, M. 2010. DMP: deterministic shared-memory multiprocessing. *IEEE Micro* 30(1):40–49.

Elmas, T.; Burnim, J.; Necula, G. C.; and Sen, K. 2013. Concurrit: a domain specific language for reproducing concurrency bugs. In *International Conference on Programming Language Design and Implementation*, 153–164. ACM.

Farzan, A.; Holzer, A.; Razavi, N.; and Veith, H. 2013. Con2colic testing. In *26th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 37–47. ACM.

Huang, J.; Zhang, C.; and Dolby, J. 2013. CLAP: recording local executions to reproduce concurrency failures. In *34th International Conference on Programming Language Design and Implementation*, 141–152. ACM.

Huang, J. 2015. Stateless model checking concurrent programs with maximal causality reduction. In *36th International Conference on Programming Language Design and Implementation*, 165–174. ACM.

Jose, M., and Majumdar, R. 2011. Cause clue clauses: Error localization using maximum satisfiability. In *32nd International Conference on Programming Language Design and Implementation*, 437–446. ACM.

Khoshnood, S.; Kusano, M.; and Wang, C. 2015. ConcBugAssist: constraint solving for diagnosis and repair of concurrency bugs. In *International Symposium on Software Testing and Analysis*, 165–176. ACM.

Lee, D.; Said, M.; Narayanasamy, S.; Yang, Z.; and Pereira, C. 2009. Offline symbolic analysis for multi-processor execution replay. In *42nd International Symposium on Microarchitecture*, 564–575. ACM.

Li, C. M., and Manyà, F. 2009. Maxsat, hard and soft constraints. In *Handbook of Satisfiability*. 613–631.

Lu, S.; Tucek, J.; Qin, F.; and Zhou, Y. 2006. AVIO: detecting atomicity violations via access interleaving invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 37–48. ACM.

Lu, S.; Park, S.; Seo, E.; and Zhou, Y. 2008. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 329–339. ACM.

Machado, N.; Lucia, B.; and Rodrigues, L. 2015. Concurrency debugging with differential schedule projections. In *36th International Conference on Programming Language Design and Implementation*, 586–595. ACM.

Machado, N.; Lucia, B.; and Rodrigues, L. 2016. Production-guided concurrency debugging. In *21st Symposium on Principles and Practice of Parallel Programming*, 29:1–29:12. ACM.

Marques-Silva, J.; Argelich, J.; Graça, A.; and Lynce, I. 2011. Boolean lexicographic optimization: algorithms & applications. *Annals of Mathematics and Artificial Intelligence* 62(3-4):317–343.

Musuvathi, M.; Qadeer, S.; Ball, T.; Basler, G.; Nainar, P. A.; and Neamtiu, I. 2008. Finding and reproducing heisenbugs in concurrent programs. In *Symposium on Operating Systems Design and Implementation*, 267–280.

Olszewski, M.; Ansel, J.; and Amarasinghe, S. P. 2009. Kendo: efficient deterministic multithreading in software. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 97–108.

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

Sebastiani, R. 2007. Lazy satisability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation* 3(3-4):141–224.