Verifying Synchronization for Atomicity Violation Fixing

Qingkai Shi, Jeff Huang, *Member, IEEE*, Zhenyu Chen, *Member, IEEE*, and Baowen Xu, *Member, IEEE*

Abstract—Atomicity is a fundamental property to guarantee the isolation of a work unit (*i.e.*, a sequence of related events in a thread) from concurrent threads. However, ensuring atomicity is often very challenging due to complex thread interactions. We present an approach to help developers verify whether such work units, which have triggered bugs due to certain violations of atomicity, are sufficiently synchronized or not by locks introduced for fixing the bugs. A key feature of our approach is that it combines the fortes of both bug-driven and change-aware techniques, which enables it to effectively verify synchronizations by testing only a minimal set of suspicious atomicity violations without any knowledge on the to-be-isolated work units, thus being more efficient and practical than other approaches. Besides, unlike existing approaches, our approach effectively utilizes all the inferred execution traces even they may not be completely feasible, such that the verification algorithm can converge much faster. We demonstrate via extensive evaluation that our approach is much more effective and efficient than the state-of-the-arts. Besides, we show that although there have existed sound automatic fixing techniques for atomicity violations, our approach is still necessary and useful for quality assurance of concurrent programs, because the assumption behind our approach is much weaker. We have also investigated one of the largest bug databases and found that insufficient synchronizations are common and difficult to be found in software development.

Index Terms—Atomicity violations, insufficient synchronization, fix, dynamic trace analysis, maximal sound verification

1 Introduction

Atomicity is a guarantee of the isolation of a work unit, which is a sequence of related events in a thread, from other concurrently executing threads. Synchronizations are commonly used for achieving atomicity [1], [2], [3], but are very challenging to be placed sufficiently [4]. Our investigation in one of the largest bug databases, Apache Jira,[1] shows that 26.3 percent problematic synchronizations are insufficient synchronizations; due to non-determinism, 70.0 percent of them cannot be found within a year after they were first introduced into the program (Section 6.5). Figs. 1 and 2 present two typical insufficient synchronizations in real world programs. In Fig. 1, developers try to use synchronization to eliminate a multi-variable atomicity violation, but a critical event in the to-be-isolated work unit is excluded from the critical section. In Fig. 2, although the work units are synchronized by locks, the lock instances are not equivalent at runtime.

Previous research has proposed many bug detection techniques [5], [6], [7], [8], [9], [10], [11], [12] to combat atomicity violations. These techniques are *bug-driven*, using bug patterns to recognize all potential atomicity violations, and thus can determine whether an atomicity violation still exists after developers fix the program by synchronization, thereby indirectly verifying whether the newly introduced synchronizations are sufficient or not. However, these techniques usually report a large number of atomicity violations including false alarms,[2] and it is hard for developers to understand which are harmful and thus should be eliminated [13]. On the other hand, recent *change-aware* techniques a.k.a. incremental testing techniques [14], [15], [16] cannot provide any guarantees for verifying synchronization, and even may miss insufficient synchronizations (i.e., cause false negatives). That is because they are not aware of the bug-patterns, but only target at code changes (including fixes) which may be irrelevant to bugs.[3]

To overcome the weakness of the state-of-the-arts, we propose a new approach to help developers determine whether some work units, which should be atomic but are not and have triggered bugs due to certain atomicity violations (with single variable or multi-variables), are synchronized sufficiently or not based on dynamic execution traces. A key observation behind our approach is that when developers encounter a buggy execution containing atomicity violations, but do not eliminate them by sufficient synchronizations, then some buggy schedule fragments that violate atomicity in the execution will still exist. Thus, like the traditional bug-driven techniques discussed above, we can make use of the buggy execution to compute all the possible

1. https://issues.apache.org/jira

- *Q. Shi, Z. Chen, and B. Xu are with the State Key Lab. for Novel Software Technology, Nanjing University, Nanjing, China. E-mail: sqk08@software.nju.edu.cn, {zychen, bwxu}@nju.edu.cn.*
- *J. Huang is with Texas A&M University, College Station, TX 77843. E-mail: jeff@cse.tamu.edu.*

Manuscript received 30 Aug. 2014; revised 23 June 2015; accepted 5 Sept. 2015. Date of publication 9 Sept. 2015; date of current version 21 Mar. 2016. Recommended for acceptance by T. Bultan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2477820

2. For example, over 1,300 suspicious atomicity violations were reported for the program Jigsaw in [12].

3. Ref. [13] reported that developers usually do not fully understand a bug before fixing, for example, 27 percent of the incorrect fixes are made by developers who never touch the source code.

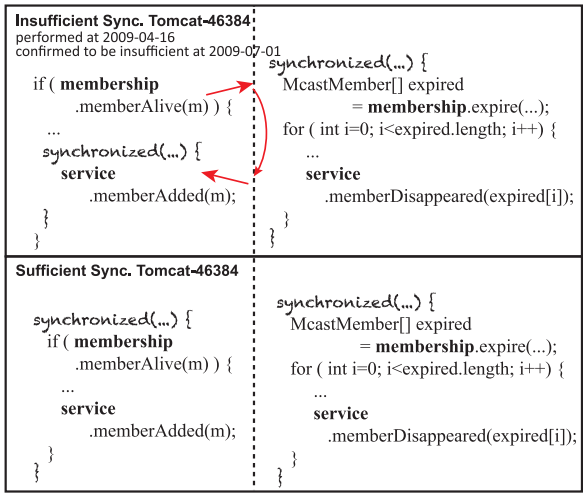


Fig. 1. Tomcat Bug-46384. The atomicity violation leads to an inconsistency between `membership` and `service`. Developers initially committed an insufficient fix (see the arrows in the top), which was not discovered until three months later. The correct fix is shown in the bottom of the figure.

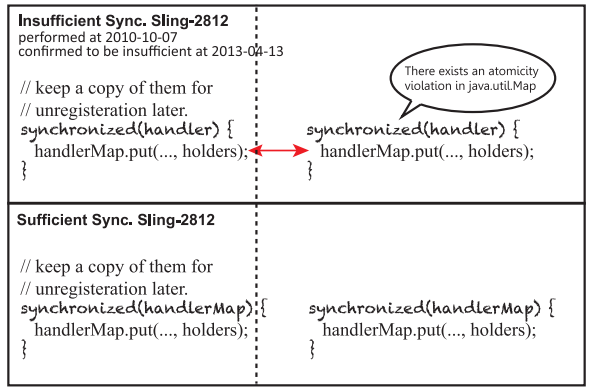


Fig. 2. SLING Bug-2812. The insufficient synchronization using a object field (`handler`), which is always different for different objects, to synchronize the codes. It will make the global object (`handlerMap`) broken.

buggy schedule fragments according to a complete set of atomicity violation patterns [4], and test them against the legal executions of the synchronized program, thereby verifying whether the synchronizations are sufficient or not. Besides, our approach is also change-aware, which can significantly reduce the number of buggy schedule fragments to test.

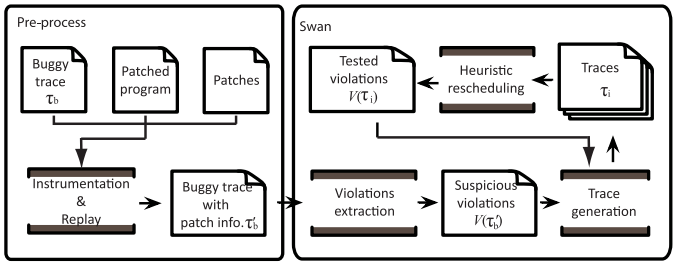
In our approach, taking advantage of existing SMT solvers [17], we encode the buggy execution as well as the patterns of atomicity violations as constraints to compute a minimal set of the schedule fragments that may violate atomicity of the work units. This is different from the traditional bug-driven techniques that need to report all such schedule fragments [6], [7], [8], [12]; thus our approach can be more efficient for verifying synchronization. To ease the presentation, we will call such schedule fragments as *suspicious violations* in the paper. We then generate new traces for the synchronized program, also using the SMT solver, under the constraints of *must-happen-before* and *lock-mutual-exclusion* [18], to test the computed suspicious violations. To make our algorithm converge quickly and more effective, we try to test a maximal number of suspicious violations every time we generate a new trace, and fully utilize the generated traces, even they might not be completely feasible. Underpinned by a sound and maximal theoretical model and several effective optimizations, our approach achieves several novel features:

(1) It can effectively verify synchronization without any knowledge on the to-be-isolated work units whose atomicity property is violated in the observed buggy execution. This is important, because developers usually do not have enough knowledge about the bug before they synchronize the program [13].
(2) It only tests a minimal set of suspicious violations to verify synchronizations, thus being very efficient (Section 3.3.1).
(3) It does not report false positive. And it does not miss any insufficient synchronization, as long as there exists a feasible trace, which can be generated based on the input trace, can manifest it (Section 3.3.2).
(4) It can dramatically speed up the verification process with the strategies that ① group and prioritize suspicious violations (Section 4.1), and ② reschedule generated traces heuristically (Section 4.2).

We anticipate three typical application scenarios of our approach. First, during in-house development, when an execution fails due to atomicity violations, developers would fix their programs with synchronizations. And then our approach can be used with the failed execution trace directly for verifying if the synchronizations for fix are sufficient. Second, when a bug is reported by the users to the bug database, and the developer would fix it with synchronization. Before fixing, developers usually need to reproduce the bug to confirm that it is a real bug. Therefore, since reproducing a bug is a pre-condition of fixing, we must have had the original buggy trace before verifying synchronization. The third application scenario is to verify an existing synchronization. This can be done by firstly removing the existing synchronizations to create an artificial bug. Then we can consider the original synchronizations as fixes for the artificial bug, and verify the synchronizations using our approach. Recently, there have existed a lot of research on bug reproduction through record and replay, such as [19], [20], [21], [22]. These techniques make it possible to record buggy executions in a compact form and with low runtime overhead. For long-running programs, we can break up the execution so that each execution segment has a tractable size. In our empirical study, we used the recent lightweight record/replay technique LEAP [19] to record the input buggy executions.

We have implemented our technique in a prototype tool, SWAN,[4] for Java programs, and evaluated it on a range of large complex multithreaded applications and compared it with two state-of-the-art techniques [5], [14]. The evaluation results show that SWAN is able to detect real insufficient synchronizations using three generated traces on average (*effectiveness*), and it is much more effective and efficient than the other techniques (*progressiveness*). Even when there are hundreds of threads, insufficient synchronization can also be

4. SWAN is the acronym of "Synchronization Was A Nightmare".



Pre-process:

```
$java -jar swan.jar --transform example.Example   \# the program to transform
                    --patch :22,:26               \# line numbers where sync. is inserted

$java -jar swan.jar --replay-record               \# indicate which phase
                    --test-case "example.Example" \# how to execute the program under test
                    --trace buggytrace            \# the input buggy trace τb
                    --output buggytrace_patchinfo \# output trace τb'
```

Swan:

```
$java -jar swan.jar --generate   --trace buggytrace_patchinfo

$java -jar swan.jar --reschedule   --test-case "example.Example"   --trace newtrace
```

Fig. 3. The framework and usage of SWAN.

detected using less than 10 generated traces (*scalability*). More importantly, SWAN can help improve the quality of synchronizations introduced by recent automatic fixing techniques that become unsound if developers cannot understand the root cause of a bug before fixing (*necessity*). In addition, our empirical study on Apache Jira shows that insufficient synchronization is common and is difficult to be found during in-house development (*potentiality*). We highlight our contributions as follows:

(1) We present a sound and maximal constraint-based model to verify synchronizations, by testing a minimal set of suspicious violations, for work units with single- or multi-variable atomicity violations without the strong assumption about the knowledge of to-be-isolated work units (Section 3).
(2) We present two optimizations to make our approach more effective and scalable (Section 4).
(3) We implement and evaluate SWAN with several large programs from popular open source projects. The results demonstrate the effectiveness, progressiveness, scalability, and the necessity of our approach. Moreover, our empirical study suggests that developers are badly in need of such an approach (Section 6).

2 OVERVIEW

We first present an overview of SWAN (Fig. 3) using a simple example (Fig. 4). In the example, two work units, $u = \langle e_1, e_2, e_4, e_5\rangle$ and $u' = \langle e_8\rangle$, should be synchronized to enforce atomicity. Suppose that developers encounter a buggy execution (modeled as a trace), $\tau_b = \langle e_1, e_2, e_4, \underline{e_8}, e_5\rangle$, in which a remote write event e_8 sets the shared variable to `Null`, and causes the program to throw a `NullPointerException` at the local event e_5. However, developers do not synchronize the work units sufficiently, excluding two events e_1 and e_2 of the work unit u from the critical section (see the comments in Fig. 4). Note that the event e_1 and e_2 can also cause atomicity violations, e.g., $\langle e_1, \underline{e_8}, e_2\rangle$.

Given the original buggy trace and the synchronization information provided in the patches, in the pre-processing phase (see the pre-processing phase in Fig. 3), we firstly replay the original buggy execution on the patched program to record a copy of the input buggy trace, which will also contain the patched synchronization information, i.e., where the patched synchronizations are performed, and what the locking objects are at runtime. To ensure successful replay, when encountering the patched synchronizations, we let the program skip them, but still record the synchronization events into the trace. In this example, the trace we record in the pre-processing phase is $\tau_b' = \langle e_1, e_2, e_3^a, e_4, e_7^{a'}, e_8, e_5, e_9^{r'}, e_6^r\rangle$, in which superscripts a/r and a'/r' are used to label the two new pairs of lock acquiring and releasing events. Compared to the original buggy trace τ_b, it has only four more synchronization events. However, τ_b' is invalid for the patched program, because the critical sections $\langle e_3^a, e_4, e_5, e_6^r\rangle$ and $\langle e_7^{a'}, e_8, e_5, e_9^{r'}\rangle$ are not mutually exclusive. Our approach starts from the trace τ_b', and reorders the events in it to generate valid traces that contain suspicious violations to verify if the patched synchronizations are sufficient. This pre-processing phase is straightforward, and will not be repeated in the following sections.

We next extract the suspicious violations from τ_b' according to the atomicity violation patterns [4]. We extract suspicious violations from τ_b' other than τ_b, because τ_b' contains the information of the patched synchronizations, which will help eliminate invalid suspicious violations. Let O_i denote the order of e_i in the trace, T_i the thread of e_i, M_i the memory accessed by e_i, and A_i the access type; then we can encode the pre-processed buggy trace τ_b' as constraint $\Phi_{\tau_b'}$, i.e.,

$$\Phi_{\tau_b'} = \bigwedge \begin{cases} O_1 < O_8 \wedge O_2 < O_8, \\ (O_4 < O_8 \wedge O_5 < O_8) \vee (O_8 < O_4 \wedge O_8 < O_5), \\ A_2 = A_4 = A_5 = read \wedge A_1 = A_8 = write, \\ M_1 = M_2 = M_4 = M_5 = M_8, \\ T_1 = T_2 = T_4 = T_5 \neq T_8, \end{cases}$$

which includes the orders, threads, accessed memories, and access types of all *read* and *write* events. Note that the critical sections $\langle e_3^a, e_4, e_5, e_6^r\rangle$ and $\langle e_7^{a'}, e_8, e_9^{r'}\rangle$ are not mutually exclusive in τ_b'. For such interactive critical sections,[5] we explore two possible order relations between them, i.e., $(O_4 < O_8 \wedge O_5 < O_8) \vee (O_8 < O_4 \wedge O_8 < O_5)$. For other events that may form high-level data races,[6] we will only explore their

```
e1 obj = ...
e2 if (obj != null) {
e3     // fix by acquiring lock l
e4     obj.functionOne();
e5     obj.functionTwo();
e6     // fix by releasing lock l
   }

e7 // fix by acquiring lock l
e8 obj = null;
e9 // fix by releasing lock l
```

Fig. 4. A simple example with atomicity violations. The original buggy trace is indicated by the arrows.

5. We will give a precise definition of interactive critical sections in Definition 2 in the next section.

6. Events that may form high-level data races means the event pairs that belong to different threads, but access to the same memory location and one of the accesses is write. These event pairs can be obtained by simply traversing the trace instead of using a complex race detector.

order relations that have existed in τ'_b. For example, only $O_1 < O_8$ is contained in $\Phi_{\tau'_b}$, because $\langle e_1, e_8\rangle \sqsubseteq \tau'_b$, and e_1, e_8 do not belong to interactive critical sections. This decision is made based on the intuition that if a bug-triggering atomicity violation is not sufficiently synchronized, it will still exist in certain executions of the patched program. As an example that will be shown subsequently, $\langle e_2, e_8, e_5\rangle$ is a bug-triggering atomicity violations in τ_b and is not synchronized sufficiently. Thus, it will still exist in certain executions of the patched program.

We then encode the patterns of single-variable atomicity violation ($\Phi_{pattern}$) using three variable events e_x, e_y, e_z, as well as constraints between them. Similarly, for multi-variable atomicity violations, we can use four or more variable events. To ease the presentation, here we only consider single-variable atomicity violations

$$\Phi_{pattern} = \bigwedge \begin{cases} O_x < O_y < O_z \\ (A_x = read \wedge A_y = write \wedge A_z = read) \vee \cdots \\ M_x = M_y = M_z \wedge T_x = T_z \neq T_y \\ e_x, e_y, e_z \in \tau'_b. \end{cases}$$

As a whole, we can solve the constraints, $\Phi_{\tau'_b} \wedge \Phi_{pattern}$, to get the suspicious violations. The solutions of the constraints are the values of O_i, T_i, M_i and A_i for all events in τ'_b and all variable events. One of the solutions for the variable events is $\{e_x = e_2, e_y = e_8, e_z = e_5\}$, meaning that the schedule fragment $\langle e_2, e_8, e_5\rangle$ may violate atomicity.

Then we can generate traces to test the extracted suspicious violation based on its order constraint (i.e., $\Phi_{\langle e_2,e_8,e_5\rangle} = O_2 < O_8 < O_5$), as well as the *lock-mutual-exclusion* constraints (Φ_{lock}) that require critical sections protected by the same lock mutually exclusive, and the *must-happen-before* constraints (Φ_{mhb}) that enforce those must-be-satisfied order relations. That is, we will solve the following constraints ($\Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi_{\langle e_2,e_8,e_5\rangle}$), and generate new traces by sorting events based on the solution (i.e., the values of order variables)

$$\begin{aligned} \Phi_{mhb} &= O_1 < O_2 < O_3^a < O_4 < O_5 < O_6^r \wedge O_7^{a'} \\ &\quad < O_8 < O_9^{r'}, \\ \Phi_{lock} &= O_9^{r'} < O_3^a \vee O_6^r < O_7^{a'}, \\ \Phi_{\langle e_2,e_8,e_5\rangle} &= O_2 < O_8 < O_5. \end{aligned}$$

If the above constraints have no solution, we can confirm that the suspicious violation $\langle e_2, e_8, e_5\rangle$ has been eliminated w.r.t. the input trace. Otherwise, we will rerun the program following the generated trace to validate the suspicious violation.[7] The generated new trace for the example is $\tau = \langle e_1, e_2, e_7^{a'}, e_8, e_9^{r'}, e_3^a, e_4, e_5, e_6^r\rangle$. Clearly, it is feasible and rescheduling it will cause the failure again, which indicates that the synchronization is insufficient.

Note that, when extracting suspicious violations, we do not get the suspicious violation $\langle e_1, e_8, e_2\rangle$, which can be reported by the traditional techniques that attempt to exhaustively report all suspicious violations [6], [7], [8], [12]. Although we miss such suspicious violations, the suspicious violations extracted by our approach are sufficient, and also necessary, for verifying synchronization, no matter the missed ones are real bugs or not. Therefore, our approach is more efficient. We will prove the sufficiency and necessity in the next section.

Because new synchronizations are added into the program, in practice, not all the traces generated based on the original buggy trace are guaranteed to be feasible. This may affect the effectiveness of synchronization verification. In our practical approach, we design a heuristic strategy to fully utilize such infeasible traces to minimize the risk of false negatives. Moreover, for real world programs with a large number of threads and synchronization operations, there may exist enormous suspicious violations to test, which makes the approach hard to scale. To address such challenges, we have designed a few optimizations that make our approach practical for real world large complex programs.

In the next two sections, we first present the theoretical model of our synchronization verification technique. We then present our practical approach with the optimizations.

3 Theory

In our approach, every program execution is modeled as a trace of events, which must obey some basic constraints such as the data/control flow of the program and the synchronization semantics. We first give the definition of the problem we address in this paper and a detailed constraint modeling of our approach. We then prove the sufficiency and necessity of the extracted suspicious violations, as well as the soundness and maximality of the trace generation algorithm. Finally we discuss the theoretical complexity of our algorithm to summarize this section.

3.1 Events and Traces

Concurrent object, such as shared memory locations, locks, etc., is a data object shared by threads [23]. An event is an operation performed on such a concurrent object with a group of attributes. For clarity, we consider the following attributes for each event e_i:

- T_i: the thread e_i belongs to;
- M_i: the memory location accessed by e_i;
- A_i: the access type of e_i, which is an element in {*read*, *write*, *acquire*, *release*, *fork*(t_p, t_q), *join*(t_p, t_q)};
- S_i: the location of the instruction (that e_i corresponds to) in the source code.

In the definition above, *fork*(t_p, t_q) is the operation that forks a new thread t_p in thread t_q, and *join*(t_p, t_q) is the operation that waits for the termination of a thread t_p in thread t_q. *acquire* and *release* are two synchronization operations, corresponding to acquiring and releasing locks, respectively. In this paper, another synchronization operation *wait* is treated as two consecutive *release-acquire* events, and each *notify* event is enforced to be between the two consecutive *release-acquire* events, and *notifyAll* is considered as multiple *notify* events.

The variable S_i is used to map a concurrency bug report to the events. It will also be used in one of our optimization approaches described in Section 4.1.

7. We will further discuss the issue in Section 5.2.

Besides, we associate each event e_i with an order variable:

- O_i: the order of the event e_i in the to-be-computed schedule or schedule fragment.

For example, $O_1 < O_2$ means that event e_1 should be scheduled before e_2, and $O_1 = O_2$ means the two events can be scheduled concurrently.

A trace is abstracted as a sequence of events, $\tau = \langle e_i \rangle$. Note that events in a trace are distinguished from each other, even though their corresponding instructions in the source codes are the same. As an example, instructions in a loop may be executed multiple times, but we associate different events to each instruction in the loop for different iterations. A legal trace (which corresponds to a consistent program execution) must satisfy two basic constraints [18]:

- *must-happen-before*: (a) If two events e_i and e_j belong to the same thread, and e_i occurs before e_j in some execution, then e_i must-happen-before e_j. (b) A *fork* event e_i must-happen-before the first event of the thread e_i forks; and the last event of a thread must-happens-before the corresponding *join* event.
- *lock-mutual-exclusion*: Two critical section protected by the same lock must be mutually exclusive at runtime. Suppose $\mathcal{L}$ is the set of locks that a trace τ contains, then we define the set of critical sections that protected by the same lock $l \in \mathcal{L}$ as $\mathcal{CS}_l = \{\tau_l = \langle e_{l,acq}, \ldots, e_{l,rel} \rangle \sqsubseteq \tau : ① M_{l,acq} = M_{l,rel} = l \in \mathcal{L}; ② A_{l,acq} = acquire \wedge A_{l,rel} = release; ③ \forall e_i, e_j \in \tau_l, T_i = T_j; ④ \forall \langle e_{l,acq}, e_i, e_{l,rel} \rangle \sqsubseteq \tau, T_i = T_{l,acq} \Rightarrow \langle e_{l,acq}, e_i, e_{l,rel} \rangle \sqsubseteq \tau_l\}$.

3.2 Problem Definition

Following previous research [4], we define an atomic set as a set of shared memories that must be accessed atomically to keep consistency. According to the definition, atomic sets should be always disjoint with each other.[8] A work unit u in a program is a sequence of events that operate on an atomic set in a thread, and should be isolated from other concurrent work units on the same atomic set. A *sufficient synchronization* guarantees the atomicity, *i.e.*, isolation, of each work unit on the same atomic set. On the other side, we say a work unit $u = \langle e_1, e_2, \ldots, e_n \rangle$ is *insufficiently synchronized*, if and only if there exists another work unit $u' = \langle e'_1, e'_2, \ldots, e'_n \rangle$ on the same atomic set such that there exists an event $e'_i \in u'$, as well as a feasible trace τ of the program such that $\langle e_1, e'_i, e_n \rangle \sqsubseteq \tau$.

Then, what we will address in the paper is the problem of synchronization verification, which is defined as follows.

Definition 1 (Synchronization verification). *Given a buggy trace which violates the atomicity of some* unknown *work units, and a synchronization that is expected to enforce their atomicity, the synchronization verification problem in this paper is to verify whether these work units are* sufficiently synchronized *in sequential consistency memory model.*

8. Suppose $\{a, b\}$, $\{b, c\}$ are two atomic sets, but contains the same memory location b. Because a and b should be accessed atomically, b and c should be accessed atomically, then all of them should be accessed atomically, which means they belong to the same atomic sets.

3.3 Constraint Model & Algorithm

Both of the two phases of SWAN (see Fig. 3), violation extraction and trace generation, are modeled as constraint solving problems in our approach.

3.3.1 Extracting Suspicious Violations

In the first phase, we extract a minimal set of the schedule fragments that may violate atomicity, i.e., *suspicious violations*, from the pre-processed buggy trace according to atomicity violation patterns. We encode the buggy trace τ as constraint Φ_τ, and use the variable events e_i, e_j, e_k (for single-variable) and e_i, e_j, e_k, e_l (for multi-variable) to formulate the patterns of atomicity violations as constraint $\Phi_{pattern}$. Besides, these variable events must be from τ. We then use an SMT solver to solve the conjunction of these constraints to get all single- and multi-variable suspicious violations, respectively:

$$\Phi_\tau \wedge \Phi_{pattern} \wedge e_i, e_j, e_k(, e_l) \in \tau. \tag{1}$$

The solutions of the constraints are the values of A_i, T_i, M_i and O_i for each event in $\tau = \langle e_1, e_2, \ldots \rangle$ and each variable event. Each solution of the constraint implies one suspicious violation. For example, for a single-variable atomicity violation pattern, suppose we get a solution that satisfies $A_i = A_1 \wedge T_i = T_1 \wedge M_i = M_1 \wedge O_i = O_1$, then $e_i = e_1$. Similarly, for example, $e_j = e_2$ and $e_k = e_3$. In this case, $(\langle e_1, e_2 \rangle, \langle e_2, e_3 \rangle)$ will be a suspicious violation extracted from the trace.

Constraint of trace τ (Φ_τ). The constraint of a given trace τ contains the threads, memories, access types, and order relations of events that may form high-level data races, which are the bases of suspicious violations in τ.

Since any suspicious violation is a schedule fragment between two different threads, we can model the constraint Φ_τ as the disjunction of constraints between different threads, i.e., $\Phi_{\tau \restriction T_i, T_j}$:

$$\Phi_\tau = \bigvee_{T_i \neq T_j \wedge e_i, e_j \in \tau} \Phi_{\tau \restriction T_i, T_j}$$

in which $\Phi_{\tau \restriction T_i, T_j}$ contains two parts:

$$\Phi_{\tau \restriction T_i, T_j} = \Phi^1_{\tau \restriction T_i, T_j} \wedge \Phi^2_{\tau \restriction T_i, T_j}.$$

(A) Constraints for interactive critical sections $\Phi^1_{\tau \restriction T_i, T_j}$: Remember that the input of our approach is a buggy trace with newly-introduced synchronizations. Thus some critical sections protected by the same lock may be not mutually exclusive in the input trace. We call such critical sections *interactive critical sections*, and define it as below.

Definition 2 (Interactive critical sections). *Two critical sections $\tau_1, \tau_2 \in \mathcal{CS}_l$ are* interactive *in a trace τ iff. $\exists e_1, e_2 \in \tau_1$, $e_3 \in \tau_2 : \langle e_1, e_3, e_2 \rangle \sqsubseteq \tau$.*

To ease presentation, we denote the set of interactive critical sections protected by a lock l as $\mathcal{CS}^*_l \subseteq \mathcal{CS}_l$. For any race pair, e_i and e_j, between interactive critical sections, we consider both of the two possible order relations between them,

TABLE 1
Constraints of Single-Variable (1-5) and Multi-Variable (6-8) Atomicity Violation

ID	Order Constraints	Thread Constraints	Memory Access Constraints		Description
Memory access pair for **single-variable** atomicity violations: $(\langle e_i, e_j\rangle, \langle e_j, e_k\rangle)$					
1				$A_i = read, A_j = write, A_k = read$	Expect to get the same value but do not.
2				$A_i = write, A_j = read, A_k = write$	A temporary result between local writes is seen to other threads.
3	$O_i < O_j < O_k$	$T_i = T_k \neq T_j$	$M_i = M_j = M_k$	$A_i = write, A_j = write, A_k = read$	A local read get an unexpected remote value.
4				$A_i = read, A_j = write, A_k = write$	Remote write is lost.
5				$A_i = write, A_j = write, A_k = write$	
Memory access pair for **multi-variable** atomicity violations: $(\langle e_i, e_j\rangle, \langle e_k, e_l\rangle)$					
6			$M_i = M_j$	$A_i = A_j = A_k = A_l = write$	Inconsistent final values.
7	$O_i < O_j \wedge O_k < O_l$	$T_i = T_l \neq T_j = T_k$	$\neq$	$A_i = A_l = write, A_j = A_k = read$	Observed values of shared variables are inconsistent.
8			$M_k = M_l$	$A_i = A_l = read, A_j = A_k = write$	

i.e. $O_i < O_j$ and $O_j < O_i$. Then the constraints of these race pairs $\Phi^1_{\tau\restriction T_i,T_j}$ can be encoded as following:

$$\Phi^1_{\tau\restriction T_i,T_j} = \bigwedge_{\substack{l\in L\\ \tau_1,\tau_2\in CS^*_l\\ \tau_1\neq\tau_2}} \left(\Phi^1_{\tau_1,\tau_2} \vee \Phi^2_{\tau_1,\tau_2}\right),$$

where τ_1 and τ_2 is a pair of interactive critical sections in threads T_i and T_j, and $\Phi^1_{\tau_1,\tau_2}$ and $\Phi^2_{\tau_1,\tau_2}$ describe the two possible orders between the two interactive critical sections:

$$\Phi^1_{\tau_1,\tau_2} = \bigwedge_{\substack{e_i\in\tau_1, e_j\in\tau_2\\ A_i=\alpha_i, A_j=\alpha_j\in\{read,write\}\\ \alpha_i=write\vee\alpha_j=write}} \begin{cases} O_i < O_j,\\ A_i = \alpha_i,\\ A_j = \alpha_j,\\ M_i = M_j,\\ T_i \neq T_j, \end{cases}$$

$$\Phi^2_{\tau_1,\tau_2} = \bigwedge_{\substack{e_i\in\tau_1, e_j\in\tau_2\\ A_i=\alpha_i, A_j=\alpha_j\in\{read,write\}\\ \alpha_i=write\vee\alpha_j=write}} \begin{cases} O_j < O_i,\\ A_i = \alpha_i,\\ A_j = \alpha_j,\\ M_i = M_j,\\ T_i \neq T_j. \end{cases}$$

(B) Constraints for other race pairs $\Phi^2_{\tau\restriction T_i,T_j}$: For events that do not belong to interactive critical sections, we only care about the order relations existing in the input trace. For example, if e_i and e_j are a race pair that does not belong to interactive critical sections and $\langle e_i, e_j\rangle \sqsubseteq \tau$, Φ_τ will only contain their order relation in τ, i.e. $O_i < O_j$, without the other possible order $O_j < O_i$. Note that even so, the extracted suspicious violations are sufficient, and also necessary, for the synchronization verification problem defined in Definition 1 (see Theorem 1). Then we can model the second part of $\Phi_{\tau\restriction T_i,T_j}$ as follows:

$$\Phi^2_{\tau\restriction T_i,T_j} = \bigwedge_{\substack{\forall l\in\mathcal{L},\tau_1,\tau_2\in\mathcal{CS}^*_l:\neg(e_i\in\tau_1\wedge e_j\in\tau_2)\\ \langle e_i,e_j\rangle\sqsubseteq\tau\\ A_i=\alpha_i, A_j=\alpha_j\in\{read,write\}\\ \alpha_i=write\vee\alpha_j=write}} \begin{cases} O_i < O_j,\\ A_i = \alpha_i,\\ A_j = \alpha_j,\\ M_i = M_j,\\ T_i \neq T_j. \end{cases}$$

In the worst case, for each pair of threads T_i and T_j the size of $\Phi_{\tau\restriction T_i,T_j}$ is quadratic in the size of events in the threads.

Atomicity violation patterns ($\Phi_{pattern}$). An atomicity violation (involving one or more variables) happens when an unserializable schedule breaks the atomicity property of some work units. Such unserializable schedules must satisfy one of the constraints in Table 1. These constraints correspond to the set of atomicity violation patterns defined in [4], which is proved to be complete. For example, the atomicity violation in Fig. 1 matches with the sixth constraint (ID=6) in Table 1.

Algorithm. Algorithm 1 shows how we extract suspicious violations from an input buggy trace based on Constraint (1). For each atomicity violation pattern (Line 2), we iteratively solve Φ (initially, it is Constraint (1)) to get suspicious violations (Lines 4-27). Every time we get a suspicious violation φ, we add a constraint to Φ to prevent obtaining the same solution (Line 24). For example, when we get a suspicious violation, $(\langle e_1, e_2\rangle, \langle e_2, e_3\rangle)$, the constraint $\neg(O_i = O_1 \wedge O_j = O_2 \wedge O_k = O_3)$ will be added to Φ to avoid duplicate solutions. Meanwhile, we also add its order constraint into a set Ψ (Line 25) for the next phase, trace generation. For instance, the order constraints of $(\langle e_1, e_2\rangle, \langle e_2, e_3\rangle)$ will be put into Ψ, i.e. $\Psi \leftarrow \Psi \cup \{O_1 < O_2 < O_3\}$.

Since we do not consider *must-happen-before* constraint in Constraint (1), Algorithm 1 may compute some *spurious suspicious violations*, which do not obey the basic constraint, but can be transformed to their valid counterparts by switching the orders of their race pairs (Lines 14-22). Fig. 5 provides an example of such spurious suspicious violations $(\langle e_3, e_4\rangle, \langle e_4, e_1\rangle)$, in which $O_3 < O_1$, but e_1 *must-happen-before* e_3. Nonetheless, it can be transformed to its valid counterpart, i.e., $(\langle e_1, e_4\rangle, \langle e_4, e_3\rangle)$, by switching the orders of both $\langle e_3, e_4\rangle$ and $\langle e_4, e_1\rangle$. Let us explain a little more about how the spurious suspicious violation is generated in our approach. First, in this example, we assume $\langle e_4, e_1\rangle$ is a subsequence of the input trace, and is not in the interactive critical sections; thus $O_4 < O_1$ is in Φ_τ. Besides, we assume e_3 and e_4 are in the interactive critical sections, and thus $O_3 < O_4 \vee O_4 < O_3$ is in Φ_τ. Then we will get one solution such that

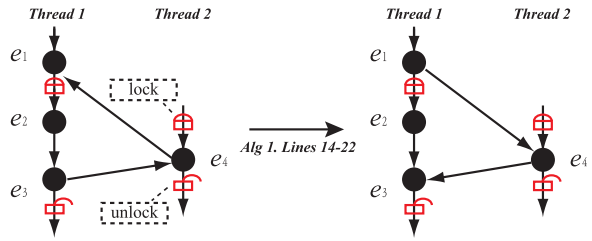


Fig. 5. A single-variable spurious suspicious violation (left), and its valid counterpart (right).

$O_3 < O_4 < O_1$, which results in the spurious suspicious violation. In fact, a spurious suspicious violation must contain a high-level data race between a pair of interactive critical sections. That is because, if newly introduced synchronizations do not lead to interactive critical sections, the input trace τ will still be feasible without any reordering and Φ_τ will only contain existing order relations in τ. Then the suspicious violations we get from Algorithm 1 must be subsequences of τ, thus obeying the *must-happen-before* constraint.

Algorithm 1. Extract Suspicious Violations

Input:
$\tau = \langle e_1, e_2, \ldots \rangle$ is the input buggy trace.
Output:
Ψ = constraint set of suspicious violations.

$\Psi \leftarrow \varnothing$;
for each $\Phi_{pattern}$ in Table 1 **do**
  $\Phi \leftarrow \Phi_\tau \wedge \Phi_{pattern} \wedge e_i, e_j, e_k(, e_l) \in \tau$;
  while Φ is solvable **do**
    Get a solution after solving Φ;
    Get the suspicious violation φ based on the solutions:
    e.g.
    $A_i = A_1 \wedge T_i = T_1 \wedge M_i = M_1 \wedge O_i = O_1 \Rightarrow e_i = e_1$
    $A_j = A_2 \wedge T_j = T_2 \wedge M_j = M_2 \wedge O_j = O_2 \Rightarrow e_j = e_2$
    $A_k = A_3 \wedge T_k = T_3 \wedge M_k = M_3 \wedge O_k = O_3 \Rightarrow e_k = e_3$
      $\Rightarrow \varphi \leftarrow (\langle e_1, e_2 \rangle, \langle e_2, e_3 \rangle)$;
        $\Phi_\varphi \leftarrow O_i = O_1 \wedge O_j = O_2 \wedge O_k = O_3$;
        $\Psi_\varphi \leftarrow O_1 < O_2 < O_3$;
    if $\varphi \leftarrow (\langle e_i, e_j \rangle, \langle e_j, e_k \rangle)$ *is a single-variable suspicious violation* **and** $O_i > O_k$ **and** e_i, e_j or e_j, e_k *are in interactive critical sections* **then**
      $\varphi \leftarrow (\langle e_k, e_j \rangle, \langle e_j, e_i \rangle)$;
    end
    if $\varphi \leftarrow (\langle e_i, e_j \rangle, \langle e_k, e_l \rangle)$ *is a multi-variable suspicious violation* **and** $O_l < O_i$ **and** e_i, e_j, e_k or e_j, e_k, e_l *are in interactive critical sections* **then**
      $\varphi \leftarrow (\langle e_j, e_i \rangle, \langle e_l, e_k \rangle)$;
    end
    if $\varphi \leftarrow (\langle e_i, e_j \rangle, \langle e_k, e_l \rangle)$ *is a multi-variable suspicious violation* **and** $O_j < O_k$ **and** e_k, e_l, e_i or e_l, e_i, e_j *are in interactive critical sections* **then**
      $\varphi \leftarrow (\langle e_j, e_i \rangle, \langle e_l, e_k \rangle)$;
    end
    if φ is not eliminated by synchronization **then**
      $\Phi \leftarrow \Phi \wedge \neg \Phi_\varphi$;
      $\Psi \leftarrow \Psi \cup \{\Psi_\varphi\}$;
    end
  end
end
return Ψ;

A difference between Algorithm 1 and traditional atomicity violation detection techniques [6], [7], [8], [12] is that the $\Phi^2_{\tau\restriction T_i,T_j}$ only cares about one of the two possible orders of a race pair, thereby reducing a large number of suspicious violations. The sufficiency and necessity of the extracted suspicious violations is proved as below, which shows that Algorithm 1 can extract a minimal set of suspicious violations for synchronization verification.

Theorem 1 (Sufficiency and necessity). *Given a suspicious violation* $\varphi = (\langle e_i, e_j \rangle, \langle e_k, e_l \rangle)$ *and its counterpart* $\varphi' = (\langle e_l, e_k \rangle, \langle e_j, e_i \rangle)$, *suppose that if* $\forall l \in \mathcal{L}, \tau_1, \tau_2 \in \mathcal{CS}_l : \neg (e_i, e_l \in \tau_1 \wedge e_k, e_j \in \tau_2)$, φ *or* φ' *must exist in some feasible traces of the program. Then suspicious violations computed by Algorithm 1 are sufficient and necessary for the synchronization verification problem defined in Definition 1.*[9]

Algorithm 1 can be considerably parallelized. That is because suspicious violations of different patterns, threads and shared memories are independent on each other, and thus we can extract suspicious violations concurrently for different patterns, threads and shared memories. The parallelization strategy enables the phase to complete in an acceptable time. We will show the empirical results in Section 6.

3.3.2 Trace Generation & Rescheduling

To verify synchronization, we then generate traces to test every suspicious violation in Ψ, with the guard of *must-happen-before* relation (Φ_{mhb}) and *lock-mutual-exclusion* condition (Φ_{lock}). Therefore, we can solve the following constraints to generate a legal trace that contains one or more suspicious violations:

$$\Phi_{mhb} \wedge \Phi_{lock} \wedge \Big(\bigwedge_{\varrho \subseteq \Psi, \varphi \in \varrho} \varphi \Big). \tag{2}$$

The solutions of the constraints are the values of order variables O_i, corresponding to all events e_i in the trace. To generate new traces, the events are reordered according to the values of order variables.

Must-happen-before constraints (Φ_{mhb}). Given a trace $\tau = \langle e_i \rangle$, according to the requirements of *must-happen-before* relation described in the previous section, Φ_{mhb} contains two parts: the program order constraint and the thread fork and join order constraint:

$$\Phi^1_{mhb} = \bigwedge_{T_i = T_j \wedge \langle e_i, e_j \rangle \sqsubseteq \tau} O_i < O_j,$$

$$\Phi^2_{mhb} = \bigwedge_{(A_i = fork(t_p,t_q) \wedge T_j = t_p) \vee (A_j = join(t_p,t_q) \wedge T_i = t_p)} O_i < O_j.$$

Although the must-happen-before relation is transitive, we need not to encode its transitivity because "$<$" is also transitive. Therefore, the size of Φ^1_{mhb} is linear in the length of τ. In most cases, we only have constant number of *fork* and *join* events. Therefore, Φ^2_{mhb} has constant-level size.

9. Please find all proofs in the paper in appendices.

Lock constraints (Φ_{lock}). The locking semantics require that critical sections protected by the same lock in τ should be mutually exclusive at runtime. That is, except the first *acquire* event, each *acquire* event must follow an *release* event on the same lock. As defined before, $\mathcal{L}$ is the set of locks, and $\mathcal{CS}_l$ is the set of critical sections that protected by the lock l, then

$$\Phi_{lock} = \bigwedge_{\substack{l\in\mathcal{L}\\ \tau_{l1},\tau_{l2}\in\mathcal{CS}_l\\ T_{l1}\neq T_{l2}.}} (O_{l1,rel} < O_{l2,acq} \vee O_{l2,rel} < O_{l1,acq})$$

Since Φ_{lock} contains every pair of critical sections that share the same lock, the size of Φ_{lock} is $O(|\mathcal{L}| \times |\mathcal{CS}_l|^2)$.

Algorithm. Algorithm 2 shows how we use Constraint (2) to verify synchronization. The objective of this algorithm is to generate a set of traces such that each suspicious violation can be tested against these traces at least once. The loop body of the algorithm (Lines 3-17) contains two main parts. In the first part (Line 3-4), we select a group of suspicious violations from Ψ, which is expected to be contained in the generated trace (Line 6), and tested during rescheduling (Line 7). In the best case, the selected group contains all the suspicious violations. In the worst case, we may need to generate traces specifically for each suspicious violation in Ψ.

Algorithm 2. Synchronization Verification

Input:
Ψ = constraint set of suspicious violations.
Output:
$\mathcal{R} \in \{\text{PASS}, \text{FAIL}\}$: FAIL indicates the sync. is not sufficient.

1 $\Phi \leftarrow \Phi_{mhb} \wedge \Phi_{lock}$;
2 **while** Φ is solvable **and** $\mathrm{P}_\Psi \neq \varnothing$ **do**
3 $\quad \mathrm{P}_\Psi \leftarrow \{\varrho \in \text{powerset}(\Psi) \setminus \{\varnothing\} : \Phi \wedge (\bigwedge_{\varphi\in\varrho} \varphi)$ is solvable$\}$;
4 $\quad \varrho_b \leftarrow$ select-and-pop-a-group-from(P_Ψ); // $\varrho_b \in P_\Psi$, Section 4.1
5 $\quad$ **repeat**
6 $\quad\quad \tau \leftarrow \text{solve}(\Phi \wedge (\bigwedge_{\varphi\in\varrho_b} \varphi))$; // generate a trace τ
7 $\quad\quad (\mathcal{R}, \varrho_b', \tau_f, \tau_t) \leftarrow$ reschedule(τ); //Alg. 3 or Alg. 4, Section 4.2
8 $\quad\quad$ **if** $\mathcal{R} = \text{FAIL}$ **then**
9 $\quad\quad\quad$ **return** FAIL;
10 $\quad\quad$ **else**
11 $\quad\quad\quad \Phi \leftarrow \Phi \wedge \neg\Phi_{\tau_f\tau_t}$; // avoid duplicate traces
12 $\quad\quad\quad \Psi \leftarrow \Psi \setminus \varrho_b'$; // remove those that have been checked
13 $\quad\quad\quad$ **if** $\varrho_b' \cap \varrho_b \neq \varnothing$ **then**
14 $\quad\quad\quad\quad$ **break**; // break to select another group
15 $\quad\quad\quad$ **end**
16 $\quad\quad$ **end**
17 $\quad$ **until** $\Phi \wedge (\bigwedge_{\varphi\in\varrho_b} \varphi)$ is unsolvable
18 **end**
19 **return** PASS;

The second part (Lines 5-17) starts by solving the conjunction of the basic constraints ($\Phi_{mhb} \wedge \Phi_{lock}$) and the constraints of the selected suspicious violations ($\bigwedge_{\varrho\subseteq\Psi,\varphi\in\varrho} \varphi$), of which the solutions are values of the order variables O_i. A new trace τ can be generated by sorting events based on the values of their order variables (Line 6). Each generated trace is rescheduled to observe whether the program will fail again (Line 7). If so, the synchronization is insufficient (Lines 8-9). Otherwise, more traces will be generated for verification.

A *basic rescheduling* method is to run the program completely based on the longest feasible[10] sub-trace τ_f of the generated trace τ ($\tau_f \sqsubseteq \tau$), and stop rescheduling once no events in τ can be scheduled. Algorithm 3 shows the *basic rescheduling* method and how it computes τ_f. In the algorithm, we always try to schedule as many events of each thread as possible. For example, when τ indicates e_i should be executed next, but the real event to execute in T_i is not e_i, we will stop rescheduling any event in T_i, because τ becomes infeasible for T_i. Algorithm 3 also outputs τ_t, which contains the remaining events (in the order in which they appeared in τ) that cannot be rescheduled.

Algorithm 3. Basic Rescheduling Method

Input:
τ : a legal trace.
Output:
$\mathcal{R} \in \{\text{PASS}, \text{FAIL}\}$: FAIL indicates the sync. is not sufficient.
ϱ_b' : suspicious violations tested at runtime.
τ_f : the longest feasible sub-trace of τ.
τ_t : the unscheduled event sequence.

1 $\tau_f \leftarrow \langle\rangle, \tau_t \leftarrow \langle\rangle, \mathcal{T} \leftarrow \varnothing, \mathcal{R} \leftarrow$ PASS;
3 **for each** $e_i \in \tau$ **do**
3 $\quad$ **if** e_i can be scheduled **and** $T_i \notin \mathcal{T}$ **then**
4 $\quad\quad \tau_f \leftarrow \tau_f e_i$; execute e_i;
5 $\quad\quad$ **if** e_i exposes the bug **then** $\mathcal{R} \leftarrow$ FAIL; **break**; **end**
6 $\quad$ **else**
7 $\quad\quad\quad \tau_t \leftarrow \tau_t e_i, \mathcal{T} \leftarrow \mathcal{T} \cup \{T_i\}$;
8 $\quad$ **end**
9 **end**
10 **return** $(\mathcal{R}, \varrho_b', \tau_f, \tau_t)$;

After rescheduling, the negated order constraint of $\tau_f\tau_t$ is added to the basic constraints to avoid generating duplicate traces (Line 11 in Algorithm 2), and the suspicious violations that have been checked are removed (Line 12 in Algorithm 2).

Theorem 2 (Soundness). *Every insufficient synchronization reported by Algorithm 2 is real.*

Theorem 3 (Maximality). *Algorithm 2 does not miss any insufficient synchronization, as long as there exists a feasible trace (that can be generated based on the input trace) for manifesting it.*

Theorem 4 (Complexity). *Let the number of thread be α, the number of shared variables β, and the number of* read *and* write *events of each shared variable in each thread N. Algorithm 2 needs to generate at most $O(2^{\alpha^2\times\beta\times N^2})$ traces to test each suspicious violation.*

The above theorems show that Algorithm 2 is sound and maximal, but may generate explosive number of traces in the worst case. In the next section (Section 4), we present the heuristics that make our approach practical for complex real world programs.

10. A feasible trace is defined as a trace that can be produced by a given program, and any prefix of a feasible trace is also a feasible trace [24].

4 Practical Approach

Firstly, we present a grouping and prioritizing strategy that enables our approach to quickly test multiple suspicious violations every time we generate a trace. We then present our re-design of the basic rescheduling method in Algorithm 3 to fully utilize all the generated traces, even though they may not be completely feasible, so that more suspicious violations can be tested each time.

4.1 Prioritizing & Grouping Suspicious Violations

Recall that we generate traces by solving Constraint (2), i.e., $\Phi_{mhb} \wedge \Phi_{lock} \wedge (\bigwedge_{\varrho \subseteq \Psi, \varphi \in \varrho} \varphi)$, which shows that we can generate a single trace to test multiple suspicious violations simultaneously. In the original algorithm (Algorithm 2), there is no strategy to guide which suspicious violations should be tested preferentially and grouped together. However, the order between suspicious violations may greatly influence the number of traces we need to generate for verification and hence impact the performance of the algorithm. For example, if we know which suspicious violation can expose the insufficient synchronization, we can generate traces for it more preferentially than others. Since such bug knowledge is usually unavailable [13], we have designed a prioritizing and grouping strategy, such that every generated trace can contain a maximal number of suspicious violations that are more likely to be harmful, thus our approach being able to expose insufficient synchronization faster and being more scalable.

The new strategy addresses this issue by assigning a priority to each suspicious violation following the rules below:

Rule 1: The suspicious violations that are more likely to be harmful should be tested preferentially.

In our approach, we extracted all suspicious violations, in which most of them are harmless, and should not be tested more preferentially than the seemingly harmful ones. Therefore, to predict harmful suspicious violations, we employ an automatic approach [25] to infer atomic sets [4], which denotes a set of shared memories that should keep consistent during some work units. The violation of the consistency property of shared memories from the same atomic set is the root cause of multi-variable atomicity violations. Clearly, the suspicious violations involving variables in the same atomic set are more likely to be harmful, and thus we assign a relatively high priority score ($\mathcal{PS}_h$) to them. Suppose the priority score of other suspicious violations is $\mathcal{PS}_l < \mathcal{PS}_h$.

Rule 2: If testing a suspicious violation implies testing more other violations, the suspicious violation should be tested preferentially.

Since the order relations between events are transitive, a suspicious violation may implies many other suspicious violations. Testing them preferentially can avoid much redundant work because we do not need to test the implied suspicious violations individually. The implication relation ($\rightarrowtail$) between suspicious violations is defined as follows.

Definition 3 (Implication relation $\rightarrowtail$). *A suspicious violation* $(\langle e_i, e_j\rangle, \langle e_k, e_l\rangle)$ implies *another one* $(\langle e_p, e_q\rangle, \langle e_r, e_s\rangle)$ *iff one of the following conditions is satisfied. $\rightarrowtail$ is a partial relation; we use $\rightarrowtail(\varphi)$ to present the set of untested suspicious violations that a suspicious violation φ implies.*

- $\langle e_i, e_j\rangle \Rightarrow \langle e_p, e_q\rangle \wedge \langle e_k, e_l\rangle \Rightarrow \langle e_r, e_s\rangle$;
- $\langle e_i, e_j\rangle \Rightarrow \langle e_r, e_s\rangle \wedge \langle e_k, e_l\rangle \Rightarrow \langle e_p, e_q\rangle$.

Here, $\langle e_i, e_j\rangle \Rightarrow \langle e_p, e_q\rangle$ iff e_p and e_i are the same event or e_p must-happen-before e_i, and meanwhile e_j and e_q are the same event or e_j must-happen-before e_q.

Based on Rule 1, we inductively define the priority of a suspicious violation φ as $\Sigma_{\varphi_i \in \rightarrowtail(\varphi)} \mathcal{PS}(\varphi_i)$, because covering φ in a generated trace means that all the suspicious violations in $\rightarrowtail(\varphi)$ will also be contained in the trace. Here, $\mathcal{PS}(\varphi_i)$ means the priority score of φ_i.

The priorities are dynamic, and can be changed depending on the rescheduling results.

Rule 3: The priority of a suspicious violation that seems infeasible should be reduced.

Even though a suspicious violation is contained in a generated trace, if it is infeasible, it cannot be tested during rescheduling. Obviously, testing an infeasible suspicious violation will waste resources. Since we cannot decide whether a suspicious violation is feasible or not in traces without running the program, every time after rescheduling, if a suspicious violation φ is not tested during rescheduling because of the infeasibility of a generated trace, the priority score will be reduced by a constant Δ_1, meaning that φ is more likely to be infeasible.

Rule 4: If a suspicious violation has been tested, the priority of other similar ones should be reduced.

In programs that frequently call libraries, or in stress testing when there are many duplicate threads, there will exist lots of suspicious violations on the same program locations. If one suspicious violation has been tested and does not trigger bugs, the other suspicious violations on the same program locations will be less likely to be harmful. As discussed in Rule 1, suspicious violations that are more seemingly harmful should be tested preferentially, which will improve the possibility of triggering bugs.

Recall that we associated another attribute S_i to each event e_i to represent its location (i.e., line numbers and source files) in the program. With this variable, we define the "similarity" relation as below.

Definition 4 (Location equivalence relation $\approx$). *A suspicious violation* $(\langle e_i, e_j\rangle, \langle e_k, e_l\rangle)$ *is* locationally-equivalent to *another one* $(\langle e_p, e_q\rangle, \langle e_r, e_s\rangle)$ *iff. one of the following conditions is satisfied. $\approx$ is an equivalence relation; we use $\approx(\varphi)$ to present the equivalence class of a suspicious violation φ.*

- $S_i = S_p \wedge S_j = S_q \wedge S_k = S_r \wedge S_l = S_s$;
- $S_i = S_r \wedge S_j = S_s \wedge S_k = S_p \wedge S_l = S_q$.

In our strategy, if any suspicious violation $\varphi_i \in \approx(\varphi)$ is tested during rescheduling, the priorities of the untested suspcious violations in $\approx(\varphi)$ will be reduced by a constant Δ_2, to indicate that a similar (i.e. locationally equivalent) one has been tested.

Effectiveness & correctness. Our strategy always prioritizes the suspicious violations that: ① are more likely to be harmful (Rule 1); ② imply more other suspicious violations (Rule 2); ③ are more likely to be feasible (Rule 3); ④ have fewer similar suspicious violations that have been tested (Rule 4). Because we do not remove any suspicious violation, our heuristic strategy does not affect the guarantees of

our approach. In our implementation, we set the parameters as $\mathcal{PS}_h = 10, \mathcal{PS}_l = 7, \Delta_1 = 1$ and $\Delta_2 = 1$. We present the empirical results in Section 6.

4.2 Heuristic Rescheduling

The basic rescheduling algorithm (Algorithm 3) only utilizes the feasible part τ_f of a generated trace τ, and discards the infeasible part τ_t similar to existing techniques [11], [12]. With such a rescheduling algorithm, the suspicious violations that contain events in τ_t will not be tested, thus limiting the effectiveness of our approach.

Our observation is that a generated trace, if it is not completely feasible, can become feasible by adding only a few events into τ_t, or removing only a few events from τ_t. That is because when fixing bugs, developers are usually very careful and only synchronize a few lines of code to avoid excessive performance degradation, thus only affecting a few local control flows [26], especially when the synchronization is insufficient or the synchronized atomicity violations are harmless.

Therefore, in this section, we look for a transformation that can transform an infeasible generated trace τ to a feasible one that contains almost all events in τ such that we can test as many suspicious violations as possible with a generated trace. However, the following theorem shows that finding such a transformation is undecidable in general.

Theorem 5 (Undecidability). *Given a trace τ of a program P and an insufficiently-synchronized version of the program P', it is undecidable to transform τ to a feasible one τ' for P' such that τ' contains the schedule fragments that violate the atomicity property of the to-be-isolated work units.*

Therefore, we in turn design the greedy rescheduling algorithm (see in Algorithm 4), which transforms the generated trace to a feasible one dynamically at runtime by adding or removing events in the generated trace. The greedy rescheduling algorithm aims to remove as few events from the generated trace as possible, so that most suspicious violations contained in the generated trace can be tested. We implement the rescheduling algorithm based on a static control flow analysis, which can determine whether an event is *reachable* from another one (Line 7) in the control flow graph. Only when an event in the generated trace cannot be scheduled and it is not reachable from any event that can be executed next, it will be removed (Line 11), because only at that time, we can confirm that the event is impossible to be scheduled in the future. If it is still reachable, we will suspend the thread and wait for the chance to schedule it (Line 7-9).

Effectiveness & correctness. Similar to other greedy algorithms, the greedy rescheduling algorithm may output a locally optimal solution which removes as few events as possible, thus testing as many suspicious violations in a generated trace as possible. As we argued before, synchronization for atomicity violations usually affect only tiny parts of the control flows in practice [26], and most parts of the generated trace are feasible for the synchronized program. Therefore, the locally optimal solution usually is also the globally optimal solution in practice, which enables us to fully utilize the generated traces, and thus being able to test far more suspicious violations than traditional techniques that discard infeasible traces [11], [12]. Since the greedy rescheduling approach only works when traces become infeasible, it not only does not affect the theoretical guarantees, but also can greatly improve the efficiency and effectiveness of our approach.

Algorithm 4. Heuristic Rescheduling Method

Input:
τ : a legal trace.
Output:
$\mathcal{R} \in \{\text{PASS}, \text{FAIL}\}$: FAIL indicates the sync. is not sufficient.
ϱ'_b : suspicious violations tested at runtime.
τ_f : the longest feasible sub-trace of τ.
τ_t : the unscheduled event sequence.

```
1  τ_f ← ⟨⟩, τ_t ← ⟨⟩, R ← PASS;
2  for each e_i ∈ τ do
3    if e_i can be scheduled then
4      τ_f ← τ_f e_i; execute e_i;
5      if e_i exposes the bug then R ← FAIL; break; end
6    else
7      E_▷ ← events that can be scheduled next;
8      if ∃e_▷ ∈ E_▷ : T_▷ = T_i ∧ reachable(e_▷, e_i) then
9        execute e_▷;
10       goto Line 3; // retry to schedule e_i
11     else
12       τ_t ← τ_t e_i;
13     end
14   end
15 end
16 return (R, ϱ'_b, τ_f, τ_t);
```

5 Discussion

In this section, we discuss the practicality of our approach.

5.1 Input Buggy Trace

Both the pattern search and the trace generation phases of our approach depend on the input buggy trace. Although we proposed heuristics to fully utilize the infeasible traces (Section 4.2), our approach is still sensitive to the original trace in theory. That is, a different input trace may exercise different program paths, which may contain events that are not synchronized. We can integrate symbolic techniques such as [27] to explore more executions to test the suspicious violations extracted from the input buggy trace. However, considering the expensive cost of symbolic techniques, they may limit the usefulness. Nevertheless, the most likely traces in practice, which can expose an insufficiently synchronized work unit, are those that are close to the input buggy one. Therefore, using the input buggy trace to generate traces allows our approach to bias the results toward real insufficient synchronization that are most likely to cause problems in practice.

5.2 Oracles

In our approach, we assume that developers can determine whether a rescheduling execution violates the atomicity of the same work units. Automatic techniques such as metamorphic approaches [28], invariant-based approach [29],

TABLE 2
Experimental Results for RQ1—Effectiveness

ID	Bug	Type†	Description	Application	LOC	#Thread	Time★ (Months)	Time with SWAN (Seconds)	#Run #Trace	Success?
1	SLING2812	MV	nonequivalent	AuthCore 1.1.0	6.5K	6	31.7	39.4	1	Y
2	Derby4723	SV	lock objects	Derby 10.5.1.1	1064K	4	54.7	90.7	1	Y
3	FOP1594	SV		Fop 0.95	186K	4	14.7	44.9	3	Y
4	Tomcat46384	MV	insufficient	Tomcat 5.5.27	535K	11	2.6	96.3	4	Y
5	Derby3308	MV	sync. with	Derby 10.2.2.0	816K	4	15.6	125	4	Y
6	Derby1573	SV	equivalent	Derby 10.2.1.6	815K	4	5.8	189	4	Y
7	Derby3909	MV	lock objects	Derby 10.2.2.0	816K	11	63.1	410	4	Y
8	Derby4124	SV		Derby 10.4.2.0	981K	8	44.5	261	4	Y

†*SV: single-variable atomicity violation. MV: multi-variable atomicity violation.*
★ *Time: the time from when an insufficient synchronization is performed to when it is found.*

etc. can help with addressing such oracle problem. In practice, this problem may not be too difficult. For example, when a Java program crashes, it will throw an exception which indicates which line threw the exception. Besides, programs usually print necessary logs in debug mode at runtime. These logs can help developers determine whether two bugs are caused due to the same reason. Moreover, even though developers cannot distinguish different buggy executions, when a program fails during rescheduling, our approach warns that there still exist bugs in the program and provides a corresponding trace, which can help developers further analyze and debug.

5.3 Deadlocks

Our approach focuses on generating and fully utilizing traces to expose insufficient synchronizations introduced by developers. Besides insufficient synchronization, developers may also introduce deadlocks during synchronization. Determining whether a fix (i.e., synchronization) introduces deadlocks or not is not the gist of our approach. In fact, detecting deadlocks has been actively studied in the literatures, e.g., [30], [31]. Our approach can integrate with any of them to help developers avoid both deadlocks and insufficient synchronizations.

6 EVALUATION

We have implemented a prototype tool, SWAN, based on Soot [32] for Java programs. Our implementation is publicly available.[11] In this section, we evaluate SWAN with the following research questions:

- **RQ1. Effectiveness.** Can SWAN expose real insufficient synchronizations?
- **RQ2. Progressiveness.** Is SWAN more effective than the state-of-the-art techniques?
- **RQ3. Scalability.** Can SWAN scale to complex executions that contain large numbers of threads?
- **RQ4. Necessity.** Considering recent automatical atomicity violation fix techniques, is SWAN still necessary and useful in practice?
- **RQ5. Potentiality.** Does insufficient synchronization commonly exist? How long does it usually take to verify a synchronization in the real world?

To perform unbiased evaluation, we evaluated SWAN using several real insufficient synchronizations from open-source Apache projects,[12] and compared with two recent techniques to show its progressiveness. For the third research question, we implemented a recent automatic fix technique for atomicity violations, and verified the synchronizations introduced by the fix technique under different assumptions. To assess the scalability of SWAN, we used the Dacapo benchmark, which contains a set of real world applications with non-trivial memory loads and has self-configured thread numbers [33]. Also, we conducted an investigation on Apache Jira, which contains bug reports ($> 200,000$) of almost all Apache projects, to illustrate the good potentiality of SWAN. All experiments are conducted on a two-core 3.07 GHz HP machine with 4 GB memory running Ubuntu 12.04.

6.1 Effectiveness

We have applied SWAN to eight real bugs caused by insufficient synchronizations (shown in Table 2). The bugs we selected contain both single-variable and multi-variable atomicity violations, and also the two cases of insufficient synchronizations illustrated in Figs. 1 and 2. At Column 8, we report the time from when an insufficient synchronization was performed to when it was found and reported (i.e. an old bug was reopened or a new bug was reported). Most of these bugs took more than one year to discover such insufficient synchronizations. However, with SWAN, developers only need three minutes on average (Column 9) to reschedule less than five traces (Column 10) to successfully verify their synchronizations (Column 11). Note that the time reported on Column 8 includes the time cost in all phases of SWAN, i.e., the time for suspicious violation extraction, trace generation, and rescheduling.

6.2 Progressiveness

To further evaluate the effectiveness of SWAN, we implemented two dynamic techniques, ASSERTFUZZER and CAPP, and compared them with SWAN. ASSERTFUZZER [5] is an active testing technique that detects both single- and multi-variable atomicity violations. CAPP [14] is a change-aware regression testing technique, which focuses on preemption prioritization and exploits code changes and their impacts.

11. http://swan.qingkaishi.com

12. http://www.apache.org

TABLE 3
Comparison

ID	SWAN		ASSERTFUZZER		CAPP	
	#Run	Success?	#Run	Success?	#Run	Success?
1	1	Y	8 (↑ 7)	Y	7 (↑ 6)	Y
2	1	Y	6 (↑ 5)	Y	10 (↑ 9)	Y
3	3	Y	9 (↑ 6)	Y	-	N
4	4	Y	-	N	9 (↑ 5)	Y
5	4	Y	18 (↑ 14)	Y	11 (↑ 7)	Y
6	4	Y	-	N	19 (↑ 15)	Y
7	4	Y	-	N	-	N
8	4	Y	13 (↑ 9)	Y	-	N

We repeated the experiment for **RQ1** using ASSERTFUZZER and CAPP. The results of this evaluation are shown in Table 3. In the table, the data in the first column are the IDs of benchmark programs, which have been shown in Table 2. The **#Run** column shows the execution times for synchronization verification, if the insufficient synchronization can be detected. The **Success?** column shows whether a technique can successfully detect an insufficient synchronization in 20 execution times. The evaluation shows that SWAN only needs to repeat executing programs at most four times to verify all these synchronizations, while ASSERTFUZZER and CAPP cannot detect half of them in 20 executions. For the cases ASSERTFUZZER and CAPP succeed, they usually need at least three times of execution times compared to SWAN.

Why SWAN is more effective? ASSERTFUZZER is a recent bug-driven technique that attempts to exhaustively test all suspicious violations using bug patterns at runtime. SWAN only tests a subset of them, actually a minimal set of suspicious violations, to verify synchronizations, and thus is more efficient. On the other hand, CAPP is a recent change-aware testing technique, which care about code changes, but is not aware of how to connect code changes with bugs. Therefore, it cannot provide any guarantees for verifying synchronization, and was less effective than SWAN.

6.3 Scalability

The scalability of SWAN depends on the number of traces that will be generated for verification. If a program needs to be rescheduled too many times to verify a synchronization, our approach will be not so interesting.

Obviously, the number of generated traces depends on the complexity of an input trace. An input trace that contains more *write* operations on shared memories has more high-level data races, thus being more difficult to verify synchronizations (Theorem 4). Hence, we select three representative programs from the Dacapo benchmark suite in the evaluation, which have different percentages of *write* operations. The three programs are (1) `Avrora`, a program that contains the highest proportion of *write* operations (70 percent *read*, 30 percent *write*) in Dacapo; (2) `Tsp`, which has the normal percentage of *read* and *write* operations (89 percent *read*, 11 percent *write*); (3) and `Moldyn`, in which almost all the operations accessing shared memory locations are *read* operations (99 percent *read*, 1 percent *write*).

We studied the number of traces generated by SWAN using 4 to 128 threads for each program. For each thread number, we first record a trace of its execution, and extracted all suspicious violations from it. We then randomly select one of the extracted suspicious violations as the target to synchronize, and insert synchronization to simulate the fixes. To avoid bias, we repeated our approach 100 times for each thread number, and calculated the average number of traces generated by SWAN.

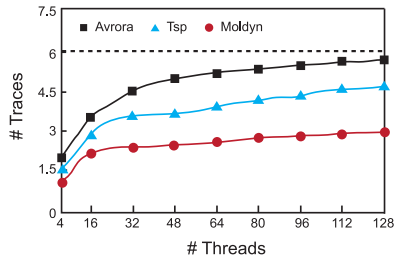


Fig. 6. Scalability. Traces versus Threads.

The results of the evaluation are shown in Fig. 6, which show that the growth rate of the number of generated traces is very slow. More importantly, the maximal number of traces generated by our algorithm does not exceed 6, which means developers can verify their synchronizations very efficiently, and our approach can scale well to programs that contain hundreds of threads.

6.4 Necessity

Recently, there have existed several automatic fixing techniques for atomicity violations, which can provide soundness guarantees [1], [2], [3]. In this case, it is a natural question that, is our approach still necessary and useful in practice? For this research question, we implemented one of the most recent fixing techniques, Axis [3] (other techniques are similar), and used SWAN to verify the synchronization introduced by Axis in two contexts using the same benchmarks described before.

In the first context, we provided Axis with the complete information about the to-be-isolated work units, including all involved variables and statements. Using SWAN, we validated the soundness of Axis, which guarantees to eliminate a *given* atomicity violation.

Considering that it is usually impossible for a developer to obtain a complete information of a bug [13], in the second context, we randomly hide some information of the to-be-isolated work units. To make the experiment more comprehensible, we conducted a case study of the bug report `FOP-1594` shown in Fig. 7. The bug contains three suspicious violations $\langle e_1, e_8, e_2\rangle$, $\langle e_2, e_8, e_4\rangle$ and $\langle e_1, e_8, e_4\rangle$ that satisfy the constraints in Table 1. Providing either the first or the second one (but not both) to Axis will lead to insufficient synchronization (Figs. 7b and 7c), because Axis can only fix the input atomicity violations but have no additional component to inspect whether the input information is complete or not. Interestingly, the insufficient synchronization introduced by Axis in Fig. 7c is the same as that in the bug report `FOP-1594`, and SWAN successfully exposed it.

In summary, recent automatic fixing techniques can guarantee to synchronize atomicity violations sufficiently, but this is true under the assumption that the input bug information is complete. This assumption is too strong in practice [13]. In contrast, SWAN is able to find all these insufficient synchronizations with a weaker and more practical

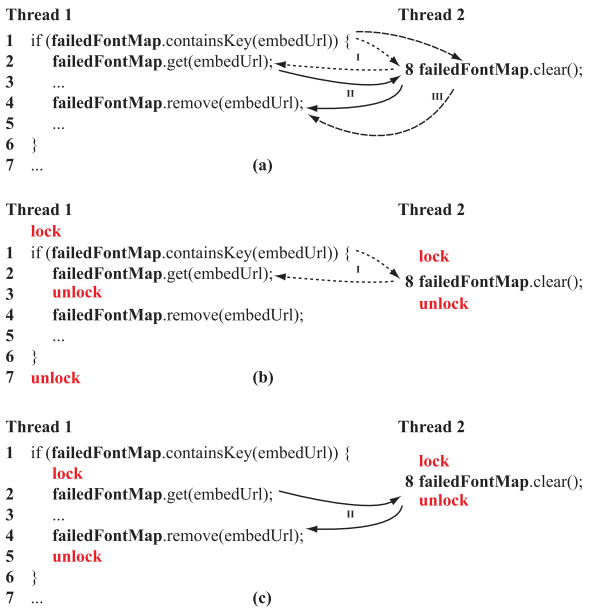


Fig. 7. Case study. (a) Three kinds of arrows indicate three possible bug-triggering atomicity violations (Table 1), in which III is the root cause. (b) The insufficient sync. introduced by Axis, when I is the input. (c) The insufficient sync. introduced by Axis, when II is the input. It is the same as that in the bug report FOP-1594.

assumption that developers are not required to point out where the bug-triggering atomicity violations locate and which variables are involved in. Therefore, our approach is still necessary and useful, and to some extent, more practical than existing automatic fixing techniques.

6.5 Potentiality

Previous work [34] has shown that about 2/3 of non-deadlock concurrency bugs are atomicity violations, which can be fixed by synchronizations. And because the developers and the code reviewers often may not have sufficient relevant knowledge on bugs, they usually cannot completely fix bugs, or even introduce new bugs [13].

To further understand the potentiality of our approach in the real world, we investigated the bug database of Apache projects, i.e., Apache Jira, which contains more than $200,000$ bugs, to study the characteristics of insufficient synchronization.

To effectively collect concurrency bugs related to synchronization, similar to the previous work [34], we used a large set of keywords like "race", "synchronization", "concurrency", "lock", "atomic" and their variations to search for related bug reports. From the thousands of bug reports that we obtained, we manually checked them and got 133 related bugs that have clear descriptions. Unfortunately, such manual work cannot be replaced by automatic techniques. To minimize subjectivity, we tried our best to conduct a double verification of the total synchronization-related bugs that we obtained by manual search.

In the 133 bug samples, 71.0 percent bugs are synchronized correctly, and the others are problematic,[13] which contains many insufficient synchronization (7.5 percent in the total, and 26.3 percent in the problematic samples). We also studied the days that developers needed to verify synchronization for atomicity violations. We found that even though some bugs were fixed correctly by synchronization, they still cost more than a month (31.6 percent) or even a year (5.3 percent) to confirm the correctness. Our investigation also reveals that it is difficult to find insufficient synchronization because insufficient synchronizations can indeed reduces the occurrence possibility of an atomicity violation. Only 30 percent of the insufficient synchronizations could be found in a year after they were introduced into the program.

In summary, insufficient synchronizations are pervasive in real world programs, and manually reasoning such insufficient synchronizations is time-consuming. An automatic synchronization verification technique like ours will be very useful for improving the effectiveness of concurrency bug fixing in practice.

13. The synchronization that may affect performance, but not correctness is not considered problematic in this paper.

7 Related Work

We summarize the related work in this section, and compare them with ours.

Fixing techniques. Synchronization is the most commonly-used method [1], [2], [3] to eliminate atomicity violations, which is an important class of concurrency errors [34]. All these fix techniques depend on the assumption that their inputs are the exact bug-triggering atomicity violations, otherwise unnecessary and insufficient synchronization will be introduced. However, although these techniques usually design strategies to avoid deadlocks, only a few of them [1], [2] use simple methods to test whether work units are sufficiently synchronized or not. Our approach has a weaker and more practical assumption that we only has a bug-triggering execution without any other knowledge about the to-be-isolated work units, which can help improve the quality of synchronizations introduced by these techniques.

Predictive techniques. A large number of techniques have been proposed for detecting or predicting concurrency bugs, including predictive trace analysis techniques [12], [35], [36], which analyze traces of a program and report suspicious read/write patterns in the program; active testing techniques [5], [10], [11], which test a program by weaving threads to expose bugs with high possibility; static analysis techniques [37], which statically analyze a whole program for bug prediction; and model checking techniques [38], [39], [40], which detects concurrency bugs by searching schedule space exhaustively with a given model. However, these techniques may waste resources on unrelated codes when used for verifying synchronization. Our approach can help developers determine whether the work units, which have triggered a bug due to atomicity violation, have been synchronized sufficiently or not by testing only a minimal set of suspicious violations, even though they do not know where the bug is [13]. In addition, unlike existing techniques, such as [11], [12], our approach fully utilizes infeasible traces, which improves both the effectiveness and efficiency.

Regression testing techniques. Regression testing techniques for concurrent programs usually take code changes into consideration like our approach. These techniques are

also known as change-aware or incremental testing techniques. Typically, such techniques include regression model checking [15] and delta execution [26], which focus on how to speed up regression testing for concurrent programs; change-aware preemption prioritization [14], which focuses on preemption prioritization by exploiting code changes and their impacts; and mutual replay techniques [16], which can allow a recorded execution of an application to be replayed with a patched version of the application, but cannot provide any guarantee for synchronization verification. The weakness of these techniques are their strong assumption on code changes. That is to say, they only focus on code changes that are assumed to be related with the bugs to fix. A recent report [13] shows that there may not exist any relation between code changes and bugs, because developers may not understand a bug before fixing. Our approach introduces bug-driven approaches into change-aware techniques, and can dramatically improve the effectiveness of synchronization verification.

Using SMT solvers. The idea using constraints and SMT solvers for concurrent program analysis has been explored before our approach. It is a widely-used method to encode traces as constraints, by solving which concurrency bugs like data races and deadlocks can be exhaustively detected [18], [36], [41], [42], [43], [44]. In addition, Cerny et al. [45] proposed semantics-preserving program transformations (like lock inserting and instruction reordering) by encoding traces as constraints, which can help developers fix concurrency bugs and improve execution efficiency. Huang et al. [22] encoded a buggy execution as constraints. Solving the constraints using SMT solvers can yield a trace that can reproduce the concurrency bug. The work in [46] encoded both control-flow and data-flow information into constraints, and generated both traces and the input data to drive concurrent program testing. In addition, Deshmukh [47], [48] proposed a static approach to encode lock orders as constraints to detect deadlocks caused by incorrect usage of libraries. Our approach is different from the above ones, because it is applied in a different application scenario, where a buggy trace and its corresponding fix information are encoded together as constraints. We solve the constraints to get a minimal set of suspicious atomicity violations, and generate new traces containing them to verify if a fix is sufficient.

8 Conclusion

We have presented a synchronization verification technique as well as a tool prototype, Swan, to help developers avoid insufficient synchronization by testing a minimal set of suspicious violations. Our technique combines the fortes of both bug-driven and change-aware techniques, which enables Swan to effectively verify synchronization with a weaker and more practical assumption than existing techniques that developers usually do not know what variables are involved in bug-triggering atomicity violations before they synchronize the program. Swan is based on a sound and maximal model and is practical through a few optimizations. Our evaluation on real world programs demonstrates the effectiveness, progressiveness, scalability, necessity and potentiality of our approach.

Appendix A
Proof of Theorem 1: Sufficiency and Necessity

Proof. (Sketch) Firstly, all suspicious violations we extract conform to the unserializable interleaving patterns in Table 1, which are proved to be complete [4]. Clearly, searching for all these patterns are both sufficient and necessary, because we do not have any knowledge about the atomicity violations in the program.

Secondly, to prove the sufficiency, we only need to prove that if two work units are not synchronized sufficiently, there must exist at least one harmful atomicity violation of the work units that can be extracted by Algorithm 1.

Suppose the bug-triggering atomicity violation in the input trace τ is $\varphi = (\langle e_i, e_j\rangle, \langle e_k, e_l\rangle)$. If it involves only a single variable, then $e_j = e_k$. Then we discuss two cases.

Case 1: φ is not eliminated by the newly-introduced synchronization. In this case, there must exist a trace of the program that can contain φ, and since $\langle e_i, e_j\rangle$, $\langle e_k, e_l\rangle \sqsubseteq \tau$, the orders $O_i < O_j$ and $O_k < O_l$ will be contained in Φ_τ; thus Algorithm 1 can extract it.

Case 2: φ is eliminated by the newly-introduced synchronization, and there exist another harmful atomicity violation $\varphi' = (\langle e_p, e_q\rangle, \langle e_r, e_s\rangle)$ that belongs to the same work units and is not eliminated. According to the patterns of atomicity violations, $\varphi'' = (\langle e_s, e_r\rangle, \langle e_q, e_p\rangle)$ must also be an atomicity violation that belongs to the same work units and is not eliminated. Suppose the interactive critical sections that eliminate φ is τ_1 and τ_2, $\tau_1, \tau_2 \in \mathcal{CS}_l^*$. In this case, one of the race pairs, $\langle e_p, e_q\rangle$ and $\langle e_r, e_s\rangle$, must belong to τ_1 and τ_2. Otherwise, φ' and φ'' do not belong to the same work units with φ.

There are five sub-cases to discuss (see Fig. 8). In Cases 2.1, 2.2 and 2.3 (Figs. 8a, 8b and 8c), e_r and e_s do not belong to the interactive critical sections, then φ' and φ'' must be a multi-variable atomicity violation because $e_p \neq e_s \wedge e_q \neq e_r$.

For Cases 2.1 and 2.2, since we explore both $O_p < O_q$ and $O_q < O_p$ in Φ_τ, then if $O_r < O_s$ in the input trace, our approach can get φ', otherwise, our approach can get φ''.

For Case 2.3, it must be $O_s < O_r$ in the input trace, otherwise τ_1 and τ_2 must be mutually exclusive in the input trace, thus contradicting to our assumption that τ_1 and τ_2 has eliminated φ. Since $O_q < O_p$ is contained in Φ_τ, our approach can get φ''.

In Cases 2.4 and 2.5, (Figs. 8d and 8e), e_r belongs to one of the interactive critical sections. Then φ' may be a single-variable atomicity violation ($e_r = e_q$) or a multi-variable atomicity violation ($e_r \neq e_q$).

For Case 2.4, if $O_r < O_s$ in the input trace, since we explore both $O_p < O_q$ in Φ_τ, we can get φ' in our approach. Otherwise, we will get φ'', which is spurious, and will be transformed to its valid counterpart φ' at Lines 14-22 in Algorithm 1. Similarly, for Case 2.5, we can get φ'' in our approach.

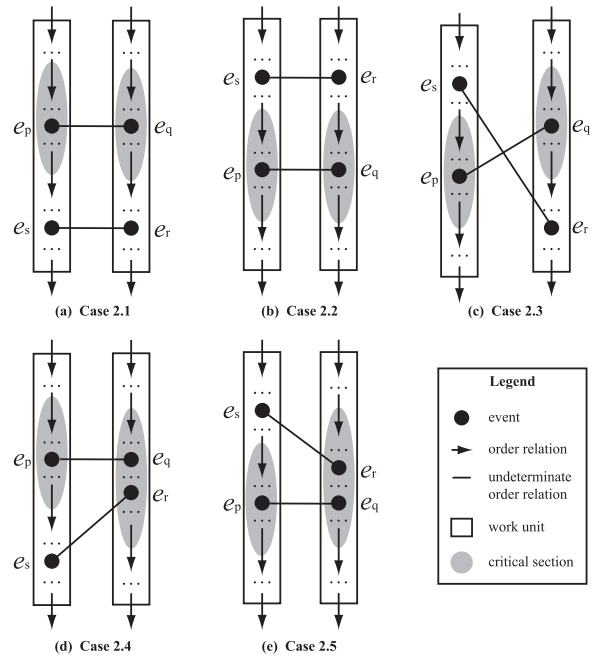


Fig. 8. Cases for proving Theorem 1. The critical sections must be interactive in the input trace, because we assume that the bug-triggering atomicity violation is eliminated by the critical sections.

Finally, we prove the necessity. Case 1 shows that the order relations in $\Phi^2_{\tau\restriction T_i,T_j}$ is necessary. Case 2 shows that order relations in $\Phi^1_{\tau\restriction T_i,T_j}$ is necessary. Since we do not have any knowledge on the to-be-isolated work units and do not know what suspicious violations are harmful, any suspicious violations composed by the two kinds of order relations are necessary for verifying synchronizations. □

Appendix B
Proof of Theorem 2: Soundness

Proof. (Sketch) Soundness is straightforward, because we reschedule the generated traces, and when the program fails, we can report a real insufficient synchronization. □

Appendix C
Proof of Theorem 3: Maximality

Proof. (Sketch) Theorem 1 has shown that the sufficiency of the extracted suspicious atomicity violations. Then the maximality is straightforward, because Algorithm 2 generates traces that satisfy Constraint (2), and only removes tested traces, and Algorithm 3 does not miss rescheduling any generated feasible trace. □

Appendix D
Proof of Theorem 4: Complexity

Proof. (Sketch) In the worst case, there exist $O(\alpha^2 \times \beta \times N^2)$ high-level data races in the trace, and we need to generate all legal traces that obey $\Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi_{\varphi}$. Since each high-level race has two possible orders to schedule, in theory, it can generate $O(2^{\alpha^2 \times \beta \times N^2})$ traces. □

Appendix E
Proof of Theorem 5: Undecidability

Proof. (Sketch) Firstly, finding such a best transformation requires running P' because predicting the executions of a program is undecidable. Suppose the introduced synchronization in P' changes some control flow in the program, which results in an infinite loop, then finding τ' is undecidable, because it is undecidable to detect an infinite loop. □

Acknowledgments

The authors wish to express deep appreciation to the anonymous reviewers for their insightful and constructive comments on an early draft of this paper. This research is supported, in part, by National Basic Research Program of China (973 Program 2014CB340702), National Natural Science Foundation of China (Grant No. 61170067, 61373013). The author, J. Huang, is partially supported by the Google Faculty Research Award. Z. Chen and B. Xu are the corresponding authors.

References

[1] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proc. 32nd ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2011, pp. 389–400.
[2] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 221–236.
[3] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 299–309.
[4] M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an object-oriented language," in *33rd ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2006, pp. 334–345.
[5] Z. Lai, S.-C. Cheung, and W. K. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 235–244.
[6] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing atomicity violation bugs from their hiding places," in *Proc.14th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2009, pp. 25–36.
[7] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting atomicity violations via access interleaving invariants," in *Proc. 12th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2006, pp. 37–48.
[8] C. Flanagan and S. N. Freund, "Atomizer: A dynamic atomicity checker for multithreaded programs," in *Proc. 31st ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2004, pp. 256–267.
[9] L. Wang and S. D. Stoller, "Accurate and efficient runtime detection of atomicity errors in concurrent programs," in *Proc. 11th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2006, pp. 137–146.
[10] C.-S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2008, pp. 135–145.
[11] F. Sorrentino, A. Farzan, and P. Madhusudan, "PENELOPE: Weaving threads to expose atomicity violations," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 37–46.
[12] J. Huang and C. Zhang, "Persuasive prediction of concurrency access anomalies," in *Int. Symp. Softw. Testing Anal.*, 2011, pp. 144–154.
[13] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram, "How do fixes become bugs?" in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 26–36.
[14] V. Jagannath, Q. Luo, and D. Marinov, "Change-aware preemption prioritization," in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 133–143.

[15] G. Yang, M. B. Dwyer, and G. Rothermel, "Regression model checking," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2009, pp. 115–124.
[16] N. Viennot, S. Nair, and J. Nieh, "Transparent mutable replay for multicore debugging and patch validation,"in *Proc. 18th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2013, pp. 127–138.
[17] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proc. 19th Int. Conf. Comput. Aided Verification*, 2007, pp. 519–531.
[18] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2014, pp. 36–47.
[19] J. Huang, P. Liu, and C. Zhang, "Leap: Lightweight deterministic multi-processor replay of concurrent Java programs," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 207–216.
[20] T. Elmas, J. Burnim, G. C. Necula, and K. Sen, "Concurrit: A domain specific language for reproducing concurrency bugs," in *Proc. 34th ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2013, pp. 153–164.
[21] J. Zhou, X. Xiao, and C. Zhang, "Stride: Search-based deterministic replay in polynomial time via bounded linkage," in *Proc. 34th Int. Conf. Softw. Eng.* 2012, 892–902.
[22] J. Huang, C. Zhang, and J. Dolby, "CLAP: Recording local executions to reproduce concurrency failures," in *Proc. 34th ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2013, pp. 141–152.
[23] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Languages Syst.*, vol. 12, no. 3, pp. 463–492, 1990.
[24] T. F. Şerbănuţă, F. Chen, and G. Roşu, "Maximal causal models for sequentially consistent systems," in *Proc. 3rd Int. Conf. Runtime Verification*, 2012, pp. 136–150.
[25] P. Liu, J. Dolby, and C. Zhang, "Finding incorrect compositions of atomicity," in *Proc. 9th Joint Meet. Found. Softw. Eng.*, 2013, pp. 158–168.
[26] J. Tucek, W. Xiong, and Y. Zhou, "Efficient online validation with delta execution," in *Proc. 14th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2009, pp. 193–204.
[27] C. Wang, R. Limaye, M. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," in *Proc. 16th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2010, vol. 6015, no. 1, pp. 328–342.
[28] H. Liu, F. Kuo, D. Towey, and T. Chen, "How effectively does metamorphic testing alleviate the oracle problem?" *IEEE Trans. Softw. Eng.*, vol. 40, no. 1, pp. 4–22, Jan. 2014.
[29] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer, "Inferring better contracts," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 191–200.
[30] Y. Cai and W. K. Chan, "MagicFuzzer: Scalable deadlock detection for large-scale applications," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 606–616.
[31] D. Marino, C. Hammer, J. Dolby, M. Vaziri, F. Tip, and J. Vitek, "Detecting deadlock in programs with data-centric synchronization," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 322–331.
[32] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *Proc. Conf. Centre Adv. Stud. Collaborative Res.*, 1999, p. 13.
[33] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proc. 21st Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst. Languages Appl.*, 2006, pp. 169–190.
[34] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proc. 13th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2008, pp. 329–339.
[35] F. Chen, T.-F. Serbanuta, and G. Rosu, "jpredictor," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008, pp. 221–230.
[36] V. Kahlon and C. Wang, "Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs," in *Proc. 22nd Int. Conf. Comput. Aided Verification*, 2010, pp. 434–449.
[37] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *Proc. Conf. Program. Language Des. Implementation*, 2003, pp. 338–349.
[38] S. Burckhardt, R. Alur, and M. M. Martin, "CheckFence: Checking consistency of concurrent data types on relaxed memory models," in *Proc. 28th ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2007, pp. 12–21.
[39] O. Shacham, M. Sagiv, and A. Schuster, "Scaling model checking of dataraces using dynamic information," *J. Parallel Distrib. Comput.*, vol. 67, no. 5, pp. 536–550, 2007.
[40] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing Heisenbugs in concurrent programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 267–280.
[41] C. Wang, M. Ganai, and A. Gupta, "Symbolic predictive analysis for concurrent programs," *Formal Aspects Comput.*, vol. 23, no. 6, p. 256, 2011.
[42] Y. Smaragdakis, J. Yi, C. Flanagan, J. Evans, and C. Sadowski, "Sound predictive race detection in polynomial time," in *Proc. 39th Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2012, pp. 387-400.
[43] A. Gupta, T. A. Henzinger, A. Radhakrishna, R. Samanta, and T. Tarrach, "Succinct representation of concurrent trace sets," in *Proc. 42nd Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2015, pp. 433–444.
[44] J. Huang, Q. Luo, and G. Rosu, "GPredict: Generic predictive concurrency analysis," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 847–857.
[45] P. Cerny, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach, "Efficient synthesis for concurrency by semantics-preserving transformations," in *Proc. 25th Int. Conf. Comput. Aided Verification*, 2013, pp. 951–967.
[46] A. Farzan, A. Holzer, N. Razavi, and H. Veith, "Con2colic testing," in *Proc. 9th Joint Meet. Found. Softw. Eng.*, 2013, pp. 37–47.
[47] J. V. Deshmukh, E. A. Emerson, and S. Sankaranarayanan, "Symbolic modular deadlock analysis," *Autom. Softw. Eng.*, vol. 18, nos. 3/4, pp. 325–362, 2011.
[48] J. Deshmukh, E. A. Emerson, and S. Sankaranarayanan, "Symbolic deadlock analysis in concurrent libraries and their clients," in *Proc. 24th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2009, pp. 480–491.

Qingkai Shi received the BEng degree from Nanjing University in June, 2012. He is a postgraduate student inthe Software Institute, Nanjing University, under the supervision of Prof. Zhenyu Chen and Prof. Baowen Xu. His research interest is program analysis and testing. He visited the Hong Kong University of Science and Technology as a visiting postgraduate student under the supervision of Prof. Charles Zhang from 2013 to 2014.

Jeff Huang received the PhD degree from the Hong Kong University of Science and Technology in 2012 and the postdoc from the University of Illinois at Urbana-Champaign in 2014. He is an assistant professor at Texas A&M University. His research focuses on developing practical techniques and tools for improving software reliability and performance. He has published extensively in premiere software engineering conferences and journals such as *TOSEM*, PLDI, OOPSLA, ICSE, FSE, ISSTA, etc. He is a member of the IEEE.

Zhenyu Chen received the bachelor's and PhD degrees in mathematics from Nanjing University. He is currently an associate professor at Software Institute, Nanjing University. He was a postdoctoral researcher at the School of Computer Science and Engineering, Southeast University, China. His research interests focus on software analysis and testing. He has about 70 publications at major venues including *TOSEM*, *TSE*, *JSS*, *SQJ*, *IJSEKE*, FSE, ISSTA, ICST, QSIC, etc. He has served as a PC co-chair of QSIC 2013, AST2013, IWPD2012, and the program committee member of many international conferences. He has won research funding from several competitive sources such as NSFC. He is a member of the IEEE.

Baowen Xu received the BS, MS, and PhD degrees in computer science from Wuhan University, Huazhong University of Science and Technology, and Beihang University, respectively. He is a professor in the Department of Computer Science and Technology, Nanjing University. His main research interests include programming languages, software testing, software maintenance, and software metrics. He has published extensively in premiere software engineering conferences and journals such as *TOSEM*, *TSE*, FSE, *JSS*, ICST, QSIC, COMPSAC, etc. He is a member of the IEEE.