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 forces 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', shows that 26.3% problematic synchronizations are insufficient synchronizations; due to non-determinism, 70.0% of them cannot be found within a year after they were first introduced into the program ([§6.5). Figures 1 and 2 present two typical insufficient synchronizations in real world programs. In Figure 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 Figure 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 includ-

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.

ing 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 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 ($\S$3.3.1).
3) It does not report false positive. And it does not miss any insufficient synchronization, as long as there exists a feasible trace, which can be generated based on the input trace, can manifest it ($\S$3.3.2).
4) It can dramatically speed up the verification process with the strategies that (1) group and prioritize suspicious violations ($\S$4.1), and (2) reschedule generated traces heuristically ($\S$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.

\(^2\) For example, over 1300 suspicious atomicity violations were reported for the program Jigsaw in [12].

\(^3\) [13] reported that developers usually do not fully understand a bug before fixing, for example, 27% of the incorrect fixes are made by developers who never touch the source code.
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, SWAN4, 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 3 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 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 (§3).

2) We present two optimizations to make our approach more effective and scalable (§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 (§6).

2 OVERVIEW

We first present an overview of SWAN (Figure 3) using a simple example (Figure 4). In the example, two work units, $u = \langle e_1, e_2, e_4, e_5 \rangle$ and $u' = \langle e_6 \rangle$, should be synchronized to enforce atomicity. Suppose that developers encounter a buggy execution (modeled as a trace), $\tau_b = \langle e_1, e_2, e_4, e_5, e_6 \rangle$, 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 Figure 4). Note that the event $e_1$ and $e_2$ can also cause atomicity violations, e.g., $\langle e_1, e_8, e_2 \rangle$.

Given the original buggy trace and the synchronization information provided in the patches, in the pre-processing phase (see the pre-processing phase in Figure 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 $\tau'_b = \langle e_1, e_2, e_3', e_4, e_5', e_6, e_8, e_5, e_6' \rangle$, 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 $\tau_b$, it has only four more synchronization events. However, $\tau'_b$ is invalid for the patched program, because the critical

4. SWAN is the acronym of “Synchronization Was A Nightmare”.

Fig. 3. The framework and usage of SWAN.

Fig. 4. A simple example with atomicity violations. The original buggy trace is indicated by the arrows.
sections \((e_3, e_4, e_5, e_6')\) and \((e_7', e_8, e_5, e_6')\) are not mutually exclusive. Our approach starts from the trace \(\tau'_{i'}\) and reorders the events in it to generate valid traces that contain suspicious violations to verify if the patched synchronizations are sufficient. This preprocessing phase is straightforward, and will not be repeated in the following sections.

We next extract the suspicious violations from \(\tau'_{i'}\) according to the atomicity violation patterns [4]. We extract suspicious violations from \(\tau'_{i'}\) other than \(\tau_{i'}\) because \(\tau'_{i'}\) 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 \(\tau'_{i'}\) as constraint \(\Phi_{i'}\), i.e.,

\[
\Phi_{i'} = \left\{ \begin{array}{l}
O_1 < O_8 \land O_2 < O_8 \\
(O_4 < O_8 \land O_5 < O_8) \lor (O_4 < O_8 \land O_6 < O_8) \\
A_2 = A_5 = A_8 = \text{read} \land A_1 = A_6 = \text{write} \\
M_1 = M_2 = M_3 = M_5 = M_8 \\
T_1 = T_2 = T_3 = T_5 \neq T_8
\end{array} \right.,
\]

which includes the orders, threads, accessed memories, and access types of all read and write events. Note that the critical sections \((e_3', e_4, e_5, e_6')\) and \((e_7', e_8, e_6')\) are not mutually exclusive in \(\tau'_{i'}\). For such interactive critical sections\(^5\), we explore two possible order relations between them, i.e., \((O_4 < O_8 \land O_5 < O_8) \lor (O_4 < O_8 \land O_6 < O_8)\). For other events that may form high-level data races\(^6\), we will only explore their order relations that have existed in \(\tau_{i'}\). For example, only \(O_1 < O_8\) is contained in \(\Phi_{e_1'}\), because \(\langle e_1, e_8 \rangle \not\in \tau_{i'}\), 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, \(\langle e_2, e_5, e_6 \rangle\) is a bug-triggering atomicity violations in \(\tau_{i'}\) 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 (\(\Phi_{\text{pattern}}\)) 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.

\[
\Phi_{\text{pattern}} = \bigwedge \left\{ O_2 < O_8 < O_5, \quad \cdots \right. \\
A_x = \text{read} \land A_y = \text{write} \land A_z = \text{read} \lor \\
M_x = M_y = M_z = T_2 = T_8 \neq T_5 \\
e_x, e_y, e_z \in \tau_{i'} \right. \\
\]

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.

As a whole, we can solve the constraints, \(\Phi_{\tau_{i'}} \land \Phi_{\text{pattern}}\), 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 \(\tau_{i'}\) and all variable events. One of the solutions for the variable events is \(\langle e_x = e_2, e_y = e_5, e_z = e_6 \rangle\), meaning that the schedule fragment \(\langle e_2, e_5, e_6 \rangle\) may violate atomicity.

Then we can generate traces to test the extracted suspicious violation based on its order constraint (i.e., \(\Phi_{e_2,e_5,e_6} = O_2 < O_5 < O_6\)), as well as the lock-mutual-exclusion constraints (\(\Phi_{\text{lock}}\)) that require critical sections protected by the same lock mutually exclusive, and the must-happen-before constraints (\(\Phi_{\text{mhb}}\)) that enforce those must-be-satisfied order relations. That is, we will solve the following constraints (\(\Phi_{\text{mhb}} \land \Phi_{\text{lock}} \land \Phi_{\langle e_2,e_5,e_6 \rangle}\)) and generate new traces by sorting events based on the solution (i.e., the values of order variables).

\[
\Phi_{\text{mhb}} = O_1 < O_2 < O_3 < O_4 < O_5 < O_6 \land O_7 < O_8 < O_9\]

\[
\Phi_{\text{lock}} = O_5 < O_6 \land O_6 < O_7
\]

\[
\Phi_{\langle e_2,e_5,e_6 \rangle} = O_2 < O_8 < O_5
\]

If the above constraints have no solution, we can confirm that the suspicious violation \(\langle e_2,e_5,e_6 \rangle\) has been eliminated w.r.t. the input order. Otherwise, we will rerun the program following the generated trace to validate the suspicious violation\(^7\).

The generated new trace for the example is \(\tau = \langle e_1, e_2, e_3', e_8, e_5, e_6, e_4, e_5, e_6 \rangle\). 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

7. We will further discuss the issue in Section 5.2.
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 $\{\text{read}, \text{write}, \text{acquire}, \text{release}, \text{fork}(t_p, t_q), \text{join}(t_p, t_q)\}$;
- $S_i$: the location of the instruction (that $e_i$ corresponds to) in the source code.

In the definition above, $\text{fork}(t_p, t_q)$ is the operation that forks a new thread $t_p$ in thread $t_q$, and $\text{join}(t_p, t_q)$ 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.

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 $L$ is the set of locks that a trace $\tau$ contains, then we define the set of critical sections that protected by the same lock $l \in L$ as $CS_l = \{\tau_i = \langle e_l, \text{acq} \rangle, \ldots, e_l, \text{rel} \rangle \subseteq \tau : \forall M_l, \text{acq} = M_l, \text{rel} = l \in L ; \forall A_l, \text{acq} = \text{acquire} \land A_l, \text{rel} = \text{release} ; \forall e_i, e_j \in \tau_i, T_i = T_j ; \forall \langle e_l, \text{acq} \rangle, e_i, e_j, \text{rel} \rangle \subseteq \tau, T_i = T_l, \text{acq} \Rightarrow \langle e_l, \text{acq} \rangle, e_i, e_j, \text{rel} \rangle \subseteq \tau\}$. 

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 $\tau$ of the program such that $\langle e_1', e_i', e_n' \rangle \subseteq \tau$.

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 Figure 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 $\tau$ as constraint $\Phi_\tau$, 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 $\tau$. We then use an SMT solver to solve the conjunction of these constraints to get all single- and multi-variable suspicious violations, respectively:

$$\Phi_\tau \land \Phi_{pattern} \land e_{i}, e_{j}, e_{k}(e_l) \in \tau$$ (1)

The solutions of the constraints are the values of $A_i, T_i, M_i$ and $O_i$ for each event in $\tau = \langle e_1, e_2, \cdots \rangle$ 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 $A_i = A_1 \land T_i = T_1 \land M_i = M_1 \land O_i = O_1$, then $e_i = e_1$. Similarly, for example, $e_j = e_2$ and $e_k = e_3$. In this case, $\langle (e_1, e_2), (e_2, e_3) \rangle$ will be a suspicious violation extracted from the trace.

Constraint of trace $\tau$ ($\Phi_\tau$). The constraint of a given trace $\tau$ 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 $\tau$.

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

$$\Phi_\tau = \bigvee_{T_i \neq T_j \land e_i, e_j \in \tau} \Phi_{\tau|T_i, T_j}$$

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

$$\Phi_{\tau|T_i, T_j} = \Phi^1_{\tau|T_i, T_j} \land \Phi^2_{\tau|T_i, T_j}$$

(A) Constraints for interactive critical sections $\Phi^1_{\tau|T_i, T_j}$: 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 $\tau_1, \tau_2 \in CS_i$ are interactive in a trace $\tau$ if $\exists e_1, e_2 \in \tau_1, e_3 \in \tau_2 : \langle e_1, e_3, e_2 \rangle \subseteq \tau$.

To ease presentation, we denote the set of interactive critical sections protected by a lock $l$ as $CS_i^l \subseteq CS_i$. For any race pair, $e_i$ and $e_j$, between interactive critical sections, we consider both of the two possible order relations between them, i.e. $O_i < O_j$ and $O_j < O_i$. Then the constraints of these race pairs $\Phi^1_{\tau|T_i, T_j}$ can be encoded as following:

$$\Phi^1_{\tau|T_i, T_j} = \bigwedge_{1 \leq \tau_1, \tau_2 \in CS_i^l \land \tau_1 \neq \tau_2} (\Phi^1_{\tau_1, \tau_2} \lor \Phi^2_{\tau_1, \tau_2})$$

where $\tau_1$ and $\tau_2$ 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:

$$\Phi^1_{\tau_1, \tau_2} = \bigwedge_{e_i \in \tau_1, e_j \in \tau_2} \begin{cases} \displaystyle O_i < O_j \quad A_i = A_j \quad M_i = M_j \quad T_i \neq T_j \end{cases}$$

$$\Phi^2_{\tau_1, \tau_2} = \bigwedge_{e_i \in \tau_1, e_j \in \tau_2} \begin{cases} \displaystyle O_j < O_i \quad A_i = A_j \quad M_i = M_j \quad T_i \neq T_j \end{cases}$$

Table 1

<table>
<thead>
<tr>
<th>ID</th>
<th>Order Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$O_i &lt; O_j &lt; O_k$</td>
</tr>
<tr>
<td>2</td>
<td>$T_i = T_k \neq T_j$</td>
</tr>
<tr>
<td>3</td>
<td>$M_i = M_j = M_k$</td>
</tr>
<tr>
<td>4</td>
<td>$A_i = A_j = A_k$</td>
</tr>
<tr>
<td>5</td>
<td>$A_i = A_j = A_k$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Access Constraints</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_i = read, A_j = write, A_k = read$</td>
<td>Expect to get the same value but do not.</td>
</tr>
<tr>
<td>$A_i = write, A_j = read, A_k = write$</td>
<td>A temporary result between local writes is seen to other threads.</td>
</tr>
<tr>
<td>$A_i = write, A_j = write, A_k = read$</td>
<td>A local read get an unexpected remote value.</td>
</tr>
<tr>
<td>$A_i = read, A_j = write, A_k = write$</td>
<td>Remote write is lost.</td>
</tr>
</tbody>
</table>

| Memory Access pair for single-variable atomicity violations: $\langle (e_1, e_j), (e_i, e_k) \rangle$ |

| Memory Access pair for multi-variable atomicity violations: $\langle (e_1, e_j), (e_i, e_k) \rangle$ |

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_i = A_j = A_k$</td>
<td>Inconsistent final values.</td>
</tr>
<tr>
<td>$A_i = A_j = A_k$</td>
<td>Observed values of shared variables are inconsistent.</td>
</tr>
</tbody>
</table>
(B) Constraints for other race pairs $\Phi^2_{r|T_i,T_j}$: 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 $\langle e_i, e_j \rangle \not\subseteq \tau$, $\Phi_r$ will only contain their order relation in $\tau$, 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 $\Phi_{r|T_i,T_j}$ as follows:

$$\Phi^2_{r|T_i,T_j} = \bigwedge_{\forall l \in \mathcal{L}, \tau_1, \tau_2 \in \text{CS}^l_\tau \subseteq \tau, \langle e_i, e_j \rangle \not\subseteq \tau} \begin{cases} O_i < O_j \\ A_i = \alpha_i \\ A_j = \alpha_j \\ M_i = M_j \\ T_i \neq T_j \end{cases}$$

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

Atomicity violation patterns ($\Phi_{\text{pattern}}$). 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 Figure 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 (Constraint (1)) get suspicious violations (Lines 4-27). Every time we get a suspicious violation $\varphi$, we add a constraint to $\Phi$ to prevent obtaining the same solution (Line 24). For example, when we get a suspicious violation, $\langle (e_1, e_2), (e_2, e_3) \rangle$, the constraint $\neg(O_1 = O_l \land O_j = O_2 \land O_k = O_3)$ will be added to $\Phi$ to avoid duplicate solutions. Meanwhile, we also add its order constraint into a set $\Psi$ (Line 25) for the next phase, trace generation. For instance, the order constraints of $\langle (e_1, e_2), (e_2, e_3) \rangle$ will be put into $\Psi$, 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). Figure 5 provides an example of such spurious suspicious violations $\langle (e_3, e_4), (e_4, e_1) \rangle$, 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., $\langle (e_1, e_4), (e_4, e_3) \rangle$, 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 $\Phi_r$. Besides, we assume $e_3$ and $e_4$ are in the interactive critical sections, and thus $O_3 < O_4 \vee O_4 < O_1$ is in $\Phi_r$. Then we will get one solution such that $O_3 < O_4 < O_1$, 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 $\tau$ will still be feasible without any reordering and $\Phi_r$ will only contain existing order relations in $\tau$. Then the suspicious violations we get from Algorithm 1 must be subsequences of $\tau$, thus obeying the must-happen-before constraint.

A difference between Algorithm 1 and traditional atomicity violation detection techniques [6], [7], [8], [12] is that the $\Phi^2_{r|T_i,T_j}$ 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.

\textbf{Theorem 1 (Sufficiency and Necessity):} Given a suspicious violation \( \varphi = \langle (e_i, e_j), (e_k, e_l) \rangle \) and its counterpart \( \varphi' = \langle (e_i, e_k), (e_j, e_l) \rangle \), suppose that if \( \forall \varphi \in \mathcal{C} : \neg (e_i, e_j, e_k, e_l) \), \( \varphi \) or \( \varphi' \) 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 \(\S 6\).

\textbf{3.3.2 Trace Generation & Rescheduling}

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

\[ \Phi_{\text{mhb}} \land \Phi_{\text{lock}} \land \left( \bigwedge_{\varphi \in \Psi} \varphi \right) \tag{2} \]

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.

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

\[ \Phi_{\text{mhb}}^1 = \bigwedge_{T_i = T_j \land (e_i, e_j) \subseteq \tau} O_i < O_j \]

\[ \Phi_{\text{mhb}}^2 = \bigwedge_{(A_i = \text{fork}(t_p, t_q) \land T_i = t_p) \lor (A_j = \text{join}(t_p, t_q) \land T_i = t_p)} O_i < O_j \]

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

\textbf{Lock constraints (\( \Phi_{\text{lock}} \)).} The locking semantics require that critical sections protected by the same lock in \( \tau \) 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

\[ \Phi_{\text{lock}} = \bigwedge_{l \in \mathcal{L}, \tau_{11}, \tau_{12} \in \mathcal{CS}_l, \tau_{11} \neq \tau_{12}} (O_{11, \text{rel}} < O_{12, \text{acq}} \lor O_{12, \text{rel}} < O_{11, \text{acq}}) \]

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

\textbf{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 \( \Psi \), 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 \( \Psi \).

The second part (Lines 5-17) starts by solving the conjunction of the basic constraints (\( \Phi_{\text{mhb}} \land \Phi_{\text{lock}} \)) and the constraints of the selected suspicious violations \( (\bigwedge_{\varphi \in \Psi} \varphi) \), of which the solutions are values of the
Algorithm 3: Basic rescheduling method

Input:  
\( \tau \): a legal trace.

Output:  
\( R \in \{\text{PASS, FAIL}\}: \text{FAIL indicates the sync. is not sufficient.} \)
\( \phi_b \): suspicious violations tested at runtime.
\( \tau_f \): the longest feasible sub-trace of \( \tau \).
\( \tau_i \): the unscheduled event sequence.

1. \( \tau_f \leftarrow \emptyset, \tau_i \leftarrow \emptyset, T \leftarrow \emptyset, R \leftarrow \text{PASS}; \)
2. for each \( e_i \in \tau \) do
   3. \( \text{if } e_i \text{ can be scheduled and } T_i \notin T \text{ then } \)
   4. \( \tau_f \leftarrow \tau_f e_i, \text{execute } e_i; \)
   5. \( \text{else if } e_i \text{ exposes the bug then } R \leftarrow \text{FAIL}; \text{break}; \text{end} \)
   6. \( \tau_i \leftarrow \tau_i e_i, T \leftarrow T \cup \{T_i\}; \)
3. end
4. return \( (R, \phi_b, \tau_f, \tau_i) \);
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 (→) between suspicious violations is defined as follows.

Definition 3 (Implication Relation →): A suspicious violation \( \langle e_i, e_j \rangle \) implies another one \( \langle e_p, e_q \rangle \) iff one of the following conditions is satisfied. \( \rightarrow \) is a partial relation; we use \( \varphi \rightarrow \) to present the set of untested suspicious violations that a suspicious violation \( \varphi \) implies.

\[
\begin{align*}
\langle e_i, e_j \rangle &\rightarrow \langle e_p, e_q \rangle \land \langle e_k, e_l \rangle \rightarrow \langle e_r, e_s \rangle; \\
\langle e_i, e_j \rangle &\rightarrow \langle e_r, e_s \rangle \land \langle e_k, e_l \rangle \rightarrow \langle e_p, e_q \rangle.
\end{align*}
\]

Here, \( \langle e_i, e_j \rangle \rightarrow \langle e_p, e_q \rangle \) iff \( e_p \) and \( e_q \) are the same event or \( e_p \) must-happen-before \( e_q \), 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 \( \varphi \) as \( \Sigma_{\varphi, \varphi \rightarrow \varphi} \text{PS}(\varphi_i) \), because covering \( \varphi \) in a generated trace means that all the suspicious violations in \( \rightarrow \) \( \varphi \) will also be contained in the trace. Here, \( \text{PS}(\varphi_i) \) means the priority score of \( \varphi_i \).

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 \( \varphi \) is not tested during rescheduling because of the infeasibility of a generated trace, the priority score will be reduced by a constant \( \Delta_1 \), meaning that \( \varphi \) 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.

Algorithm 4: Heuristic rescheduling method

Input: 
\( \tau \): a legal trace.

Output: 
\( R \in \{\text{PASS}, \text{FAIL}\} \); FAIL indicates the sync. is not sufficient. 
\( \varphi_i \): suspicious violations tested at runtime. 
\( \tau_f \): the longest feasible sub-trace of \( \tau \). 
\( \tau_i \): the unscheduled event sequence.

\[
\begin{align*}
\text{1) } & \tau_f \leftarrow \{\} \text{; } \tau_i \leftarrow \{\} \text{; } R \leftarrow \text{PASS} ; \\
\text{2) for each } & e_i \in \tau \text{ do} \\
\text{3) if } & e_i \text{ can be scheduled then} \\
\text{4) } & \tau_f \leftarrow \tau_f \cup \{ e_i \} \\
\text{5) else if } & e_i \text{ exposes the bug then} R \leftarrow \text{FAIL} ; \text{break}; \text{end} \\
\text{6) } & E_0 \leftarrow \text{events that can be scheduled next} ; \\
\text{7) if } & \exists e_0 \in E_0, : T_{e_0} = T_\tau \land \text{reachable}(e_0, e_i) \text{ then} \\
\text{8) } & \text{execute } e_0 ; \\
\text{9) else } & \text{goto Line 3} ; \text{// retry to schedule } e_i \\
\text{10) end} \\
\text{11) end} \\
\text{12) } & \tau_i \leftarrow \tau_i \cup e_i \\
\text{13) end} \\
\text{14) end} \\
\text{15) return } (R, \varphi_i, \tau_f, \tau_i).
\end{align*}
\]

Definition 4 (Location Equivalence Relation \( \approx \)): A suspicious violation \( \langle e_i, e_j \rangle \) is locationally-equivalent to another one \( \langle e_p, e_q \rangle \) 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 \( \varphi \).

\[
\begin{align*}
\varphi_i \approx & \varphi_j \text{ if } \tau \approx \varphi_i \text{ and } \tau \approx \varphi_j \\
\varphi_i \approx & \varphi_j \text{ if } \tau \approx \varphi_i \text{ and } \tau \approx \varphi_j \\
\varphi_i \approx & \varphi_j \text{ if } \tau \approx \varphi_i \text{ and } \tau \approx \varphi_j.
\end{align*}
\]

In our strategy, if any suspicious violation \( \varphi_i \in \approx (\varphi) \) is tested during rescheduling, the priorities of the untested suspicious violations in \( \approx (\varphi) \) will be reduced by a constant \( \Delta_2 \) 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 \( \text{PS}_{\Delta_0} = 10, \text{PS}_{\Delta_1} = 7, \Delta_1 = 1 \text{ and } \Delta_2 = 1 \). We present the empirical results in Section 6.

4.2 Heuristic Rescheduling

The basic rescheduling algorithm (Algorithm 3) only utilizes the feasible part \( \tau_f \) of a generated trace \( \tau \), and discards the infeasible part \( \tau_i \) similar to existing techniques [11], [12]. With such a rescheduling algorithm, the suspicious violations that contain events in \( \tau_i \) 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 \( \tau_i \), or removing only
a few events from \( \tau_t \). 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 \( \tau \) to a feasible one that contains almost all events in \( \tau \) 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 \( \tau \) of a program \( P \) and an insufficiently-synchronized version of the program \( P' \), it is undecidable to transform \( \tau \) to a feasible one \( \tau' \) for \( P' \) such that \( \tau' \) 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 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.

## 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], etc. can help with addressing such an 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\(^\text{11}\). 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 automatic 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\(^\text{12}\), 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 2-core 3.07GHz HP machine with 4GB 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 Figures 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.

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.

---

\(^\text{11}\) http://swan.qingkaishi.com
\(^\text{12}\) http://www.apache.org
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 4 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 3 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% `read`, 30% `write`) in Dacapo; (2) **Tsp**, which has the normal percentage of `read` and `write` operations (89% `read`, 11% `write`); (3) **Moldyn**, in which almost all the operations accessing shared memory locations are `read` operations (99% `read`, 1% `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**.

The results of the evaluation are shown in Figure 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 **Pop**-1594 shown in Figure 7. The bug contains three suspicious violations `⟨e_1, e_8, e_2⟩`, `⟨e_2, e_8, e_4⟩` and `⟨e_1, e_8, e_4⟩` 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 (Figure 7 (b) and (c)), 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 Figure 7 (c) is the same as that in the bug report **Pop**-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 assumption that developers are not required to point out where the bug-triggering atomicity violations locate and which
In the 133 bug samples, 71.0% bugs are synchronized correctly, and the others are problematic, which contains many insufficient synchronization (7.5% in the total, and 26.3% 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%) or even a year (5.3%) to confirm the correctness. Our investigation also reveals that it is difficult to find insufficient synchronization because insufficient synchronizations can indeed reduce the occurrence possibility of an atomicity violation. Only 30% 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.

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

6.5 Potentially

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.

variables are involved in. Therefore, our approach is still necessary and useful, and to some extent, more practical than existing automatic fixing techniques.
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 is 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, P. 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. J. 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, J. 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 forte 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**

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

**Case 2:** $\varphi$ is eliminated by the newly-introduced synchronizations, 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, $\varphi'' = (\langle e_p, e_q \rangle, \langle e_r, e_s \rangle)$ must also be an atomicity violation that belongs to the same work units and is not
eliminated. Suppose the interactive critical sections that eliminate ψ is τ1 and τ2, τ1, τ2 ∈ CS∗. In this case, one of the race pairs, ⟨ep, eφ⟩ and ⟨er, es⟩, must belong to τ1 and τ2. Otherwise, ψ′ and ψ″ do not belong to the same work units with ψ.

There are five sub-cases to discuss (see Figure 8). In Cases 2.1, 2.2 and 2.3 (Figure 8(a), (b) and (c)), er and es do not belong to the interactive critical sections, then ψ′ and ψ″ must be a multi-variable atomicity violation because ep ̸= es ∧ eφ ̸= eφ.

For Cases 2.1 and 2.2, when we explore both Oq < Oq and Oq < Oq in Φτ, then if Oq < Oq in the input trace, our approach can get ψ′, otherwise, our approach can get ψ″.

For Case 2.3, it must be Oq < Oq in the input trace, otherwise τ1 and τ2 must be mutually exclusive in the input trace, thus contradicting to our assumption that τ1 and τ2 has eliminated ψ. Since Oq < Oq is contained in Φτ, our approach can get ψ″.

In Cases 2.4 and 2.5, (Figure 8(d) and (e)), er belongs to one of the interactive critical sections. Then ψ′ may be a single-variable atomicity violation (er = eφ) or a multi-variable atomicity violation (er ̸= eφ).

For Case 2.4, if Oq < Oq in the input trace, since we explore both Oq < Oq in Φτ, we can get ψ′ in our approach. Otherwise, we will get ψ″, which is spurious, and will be transformed to its valid counterpart ψ′ at Lines 14-22 in Algorithm 1. Similarly, for Case 2.5, we can get ψ″ in our approach.

Finally, we prove the necessity. Case 1 shows that the order relations in Φ2τ∗|Tj is necessary. Case 2 shows that order relations in Φ1τ|Tj 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(α2 × β × N2) high-level data races in the trace, and we need to generate all legal traces that obey Φmhb ∧ Φlock ∧ Φτ. Since each high-level race has two possible orders to schedule, in theory, it can generate O(2n2×β×N2) 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.

**ACKNOWLEDGMENT**

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.
Qingkai Shi is a postgraduate student in Software Institute, Nanjing University, under the supervision of Prof. Zhenyu Chen and Prof. Baowen Xu. His research interest is program analysis and testing. He got a BEng from Nanjing University in June, 2012. 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 is an Assistant Professor at Texas A&M University. He received his Ph.D from the Hong Kong University of Science and Technology in 2012 and completed a Postdoc at University of Illinois at Urbana-Champaign in 2014. His research focuses on developing practical techniques and tools for improving software reliability and performance. He has published extensively in premier software engineering conferences and journals such as TOSEM, PLDI, OOPSLA, ICSE, FSE, ISSTA, etc. He is a member of the IEEE.

Zhenyu Chen is currently an Associate Professor at Software Institute, Nanjing University. He received his bachelor and Ph.D. in Mathematics from Nanjing University. He worked as 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 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 at Nanjing University. His main research interests are 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.