Axis: Automatically Fixing Atomicity Violations through Solving Control Constraints

Peng Liu Charles Zhang

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
{*lpxz, charlesz* }*@cse.ust.hk*

***Abstract*—Atomicity, a general correctness criterion in concurrency programs, is often violated in real-world applications. The violations are difficult for developers to fix, making automatic bug fixing techniques attractive. The state of the art approach aims at automating the manual fixing process but cannot provide any theoretical reasoning and guarantees. We provide an automatic approach that applies well-studied discrete control theory to guarantee deadlocks are not introduced and maximal preservation of the concurrency of the original code. Under the hood, we reduce the problem of violation fixing to a constraint solving problem using the Petri net model. Our evaluation on 13 subjects shows that the slowdown incurred by our patches is only 40% of that of the state of the art. With the deadlock-free guarantee, our patches incur moderate overhead (around 10%), which is a worthwhile cost for safety.**

I. Introduction

For concurrent programs using shared memory, atomicity is a widely used correctness criterion. It requires an execution of one thread, which involves one or more shared memory locations, not be interrupted by other threads. In the mainstream programming practice, atomicity is enforced by locks, which is notoriously difficult to be programmed correctly. Although the violations of atomicity can be detected effectively ([6], [12], [13], [25], [28], [32]), tremendous difficulties still exist in fixing them properly by the programmers, who can easily introduce new bugs such as deadlocks or unintentional performance penalties. Studies ([1], [10], [27]) show that it often takes more than one month to fix a concurrency bug and nearly 70% of the patches are still buggy in their first release.

One of the primary reasons for this difficulty, as observed by many researchers ([18], [30], [36]), is that writing synchronization code that is both *safe* and *performant* requires the intricate non-modular reasoning about the semantics of the program. The desired rigorous inter-procedural reasoning of these types of global properties is usually the strength of the compiler-based techniques. Therefore, the research of lock allocation ([11], [15], [21]) aims at replacing the manual reasoning by automatically assigning the locking operations based on atomicity specifications. These approaches can in general guarantee the freedom of deadlocks but often result in the unsatisfactory performance ([11], [15]), due to the general conservativeness of static analysis techniques, such as the thread-escape analysis [8], the array index analysis [29], the shape analysis [14], and many others.

We believe that, for a carefully engineered concurrent system, the programmer, who really understands its semantics, is more likely to produce the high quality synchronization code that has good performance as a result of the high degree of concurrency. This is simply because performance can be easily measured in concrete test runs, whereas safety is difficult to verify due to the scheduling non-determinism. An effective way of curing atomicity violations is to maximally respect the programmers' designs of lock placement and assist them in avoiding hazardous situations such as inconsistent operations and deadlocks.

One recent work, AFix [18], is the state of the art research in this direction. The primary goal of AFix is to *automate* the *manual* bug fixing process for programmers and strengthen it with the static analysis techniques such as the path analysis, the reduction of subsumed bugs, and the merging of overlapping bugs. However, limited by the inherent nature of manual reasoning, approaches such as AFix essentially lack a unified theoretical foundation to globally reason about bug fixes together with the existing lock design. Consequently, no guarantees are provided with respect to either the safety or the performance of the patched program. Our evaluation of AFix on large real systems also shows that the AFix sometimes incurs the degraded performance and, worse, frequent deadlocks.

Different from AFix, we propose a fully automatic bug fixing technique that is systematic, rigorous, and providing both safety and performance guarantees. For the end user, our technique exhibits two distinctive and highly useful properties. First, our technique can simultaneously reason and fix any number of correlated atomicity violations involving an arbitrary number of variables. Second, we guarantee that the fix not only is deadlock-free but also incurs the *minimal interference* to the degree of concurrency of the original code.

Our technique achieves these properties by modeling the concurrent properties of a program as Petri nets [22] and using a branch of control theory called the Supervision Based on Place Invariants (SBPI) [17] as our theoretical foundation. We use novel methods to reduce the problem of fixing atomic violations to solving a set of control constraints on Petri nets. By solving the constraints with an SBPI constraint solver, we compute the locking additives to the original program, which prohibit the illegal interleavings of threads. The SBPI theory guarantees that the solution satisfies the constraints, i.e., preventing the violations from happening. It also guarantees that the solution incurs the *minimal interference*[1] to the degree of concurrency of the program, hence, maximally respects the programmer's original design. We extend the earlier application ([33], [34]) of SBPI on the deadlock avoidance and guarantee that, if the original program is deadlock-free, our solution does not cause the patched program to deadlock.

We implemented our approach as a tool, Axis, and evaluated it against AFix, the state of the art automatic approach for fixing atomicity violations, using 13 subjects including three large scale real world programs. Our evaluation shows that, for these large scale subjects, our patched version outperforms the AFix solution by 6% to 9%; the overhead of our patched version is much smaller than (only 40% of) the overhead of the AFix solution. Considering the subjects are well engineered and AFix is efficient, such improvement is significant. Equally important, our patched program scales well when the thread number increases. Besides, our patched version with the deadlock-free guarantee never incurs deadlocks in our experiments while incurring a moderate (average 10%) overhead.

We make a significant contribution to the automatic fixing of atomicity violations, which includes the following:

1) We propose an automatic approach to fix the atomicity violations, which guarantees the minimal interference to the concurrency degree of the program, and no introduction of deadlocks.
2) We reduce the fixing of atomicity violations to solving the control constraints, which brings multiple computational advantages.
3) We propose a new Petri net modeling which combines the dynamic calling context information and the static program code. The modeling is scalable, complete, and precise for the violation fixing.

The rest of the paper is organized as follows. We first briefly describe the overview of our approach in Section II. Then, we present our approach in Section III. Section IV and Section V state the implementation and evaluation, respectively.

[1]The guarantee is referred to as the *maximal permissiveness* of concurrency degree in the control theory, we may use the two terms interchangeably.

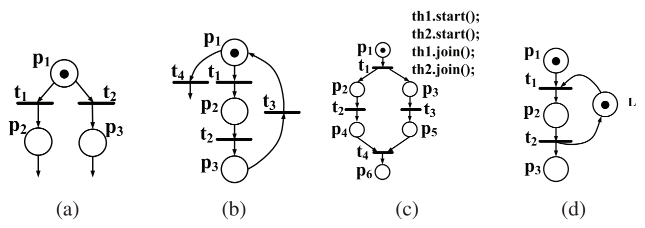


Figure 1: Basic Petri subnets. (a) Branch. (b) Loop. (c) Start and Join. (d) Lock.

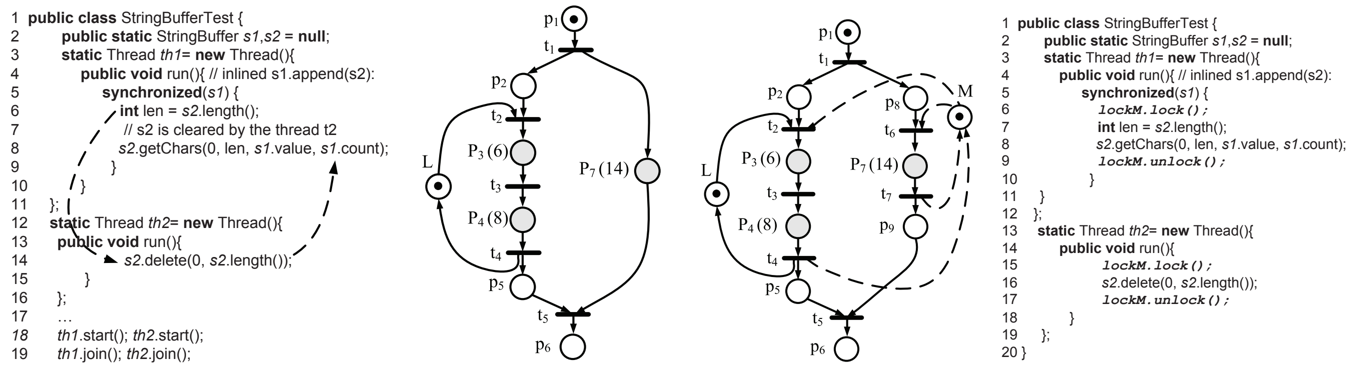
II. Overview

We present our technique in a nutshell by giving a primer on the Petri net modeling of program control flows first, followed by high level steps of our technique.

Petri Net Primer. Petri net [22] is a compact representation of state machines that avoids the state explosion problem. A comprehensive overview of Petri net models and their applications is given by Murata [22] and we briefly outline the Petri net models of the basic control flow constructs including *branch, loop, fork, join*, and *lock*, as illustrated in Figure 1. Petri net is a bipartite directed graph, which contains two types of nodes, places (circles) and transitions (horizontal bars), connected by the arcs. Each place may contain tokens (black dots), each of which marks the current position of the execution flow of a thread. Each transition can be triggered independently of each other. When being triggered, the transition removes the tokens from each of its input places and replenishes tokens to each of its output places. Each arc is associated with a weight, i.e., a number that determines the number of tokens to remove or to replenish. If any input place does not contain sufficient tokens, the transition is not triggered.

Petri net models program statements (or basic blocks) as *places* and the control flow as the *transitions*. For instance, in Figure 1(a), the branch is modeled using a single token in place p_1 that flows to either p_2 or p_3 through a simple transition (one source and one target), t_1 or t_2. As in Figure 1(c), the *start* operation is represented by the split transition (one source and two targets), t_1, removing the token in p_1 and simultaneously replenishing tokens to p_2 and p_3. Locks are modeled using two tokens, one representing the program flow and the other the availability of the lock, as illustrated in Figure 1(d). The merging transition (two sources and one target), t_1, says both tokens need to be available to enable the triggering of the transition, and the triggering of t_1 removes the tokens, simulating the *lock* operation. The splitting transition t_2 returns the token back to L, simulating the *unlock* operation.

Our Technique. We use a slightly simplified version of the well-know atomicity violation from the `StringBuffer` class in JDK 1.4 to highlight the essential steps of our technique. The interleaving sequence and the



```
1  public class StringBufferTest {
2        public static StringBuffer s1,s2 = null;
3        static Thread th1= new Thread(){
4            public void run(){ // inlined s1.append(s2):
5                synchronized(s1) {
6                    int len = s2.length();
7                     // s2 is cleared by the thread t2
8                    s2.getChars(0, len, s1.value, s1.count);
9                  }
10           }
11    };
12    static Thread th2= new Thread(){
13      public void run(){
14                s2.delete(0, s2.length());
15           }
16      };
17      …
18      th1.start(); th2.start();
19      th1.join(); th2.join();
```

```
1  public class StringBufferTest {
2        public static StringBuffer s1,s2 = null;
3        static Thread th1= new Thread(){
4            public void run(){ // inlined s1.append(s2):
5                synchronized(s1) {
6                    lockM.lock();
7                    int len = s2.length();
8                    s2.getChars(0, len, s1.value, s1.count);
9                    lockM.unlock();
10               }
11     }
12   };
13    static Thread th2= new Thread(){
14         public void run(){
15                lockM.lock();
16                s2.delete(0, s2.length());
17                lockM.unlock();
18         }
19     };
20 }
```

(a) (b) (c) (d)

Figure 2: (a) The original code. (b) The Petri net model with offending places shaded. (c) The augmented Petri net. (d) The patched code.

offending statements are indicated by the arrows in Figure 2. Given a violation report containing the offending program statements, such as line 6, 8, and 14 in our example, we first construct a Petri net model for this program, illustrated in Figure 2(b). We next encode this Petri net and the offending places (statements) as a set of *control constraints*, which can be mathematically solved to yield an augmented Petri net as shown in Figure 2(c). Our technique guarantees that, if the new lock M and the arcs (dashed lines) are added, the new program has the following properties: 1. the violation will no longer happen (*correctness*); 2. there is no deadlock between the new lock M and the existing lock L (*deadlock-freeness*); 3. the new lock is used only when the interleavings of the threads trigger the specific bug (*maximal permissiveness*). The augmented Petri net in Figure 2(c) corresponds to the patched code in Figure 2(d).

The key to achieve these salient properties is to model the absence condition of each atomicity violation as a control constraint and to satisfy the constraint by augmenting the Petri net with new lock places and new arcs. For instance, the absence condition of the exemplary violation is, line 14 cannot interleave between line 6 and line 8, which is equivalent to two *control constraints*: 1. places p_7 and p_3 cannot simultaneously contain tokens; 2. places p_7 and p_4 cannot simultaneously contain tokens. Then, we express these constraints in succinct linear inequalities (Section III-B), and use the SBPI constraint solver to compute an augmentation to the Petri net, to guarantee that all constraints are satisfied. In addition, we extend the siphon analysis of Wang et al. [34] to eliminate any possible deadlocks introduced by the augmentation. Treating atomicity violations with a constraint solving system has many computational advantages over existing approaches. The solver considers all correlated bugs and produces the optimum (*maximal permissive*) solutions, rather than simply relying on merging heuristics. The solver computes automatically, without any ad hoc treatments for loops or branches, the lock placement that guarantees to be free of bad lock practices. In addition, the mathematic form of the constraints allows further algebraic optimizations such as the rank reduction. We now describe our constraint solving approach in detail.

III. Fixing Atomicity Violations with Control Constraints

In this section, we first explain how the Petri net models and control constraints are constructed. We then explain how they are solved by the SBPI constraint solver to generate patches for fixing the violations. At last, we show the general applicability of our approach, the deadlock-free guarantee, and the guarantee of lock placement.

A. On-Demand Petri Net Construction

Our technique takes as input the reports of violations, including both single-variable and multi-variable violations. For the ease of technical discussion, we primarily discuss the single-variable violations and briefly outline later in the paper how multi-variables are treated.

The quality of the bug reports greatly affects the quality of bug fixes in general. Although our technique can be applied to statically detected violations, our current focus is to deal with dynamically discovered bugs with both the offending statements and their calling contexts available. Most dynamic bug detection tools ([16], [26]) can provide this information easily. Each single variable atomicity violation can be characterized by three statements, s_α, s_β and s_γ, where s_α and s_β are executed by the same thread and s_γ is a remote statement. Given the calling contexts of these statements, the context-sensitive Petri net can easily be constructed starting from the entry point of the thread, by constructing the subnets intra-procedurally and by linking the subnets at the call sites inter-procedurally. If the thread is created in a loop or by a recursive call, we clone the Petri net starting from the thread entry point to represent two threads. Note two threads are sufficient to express atomicity violations [19].

B. Control Constraints

Constructing the control constraints on the Petri net is a key step in fixing the violations bugs. However, the construction is not straightforward because atomicity bugs are characterized by their dynamic behaviors, whereas control constraints express the structural information of Petri nets, which is static by nature. Our transformation method is based on the following observation. For a violation, (s_α, s_β, s_γ), to occur at runtime, there must exist a statement inclusively between s_α and s_β that is executed concurrently with s_γ. This is an equivalent condition of the atomicity violation [28]. We define this observation formally as Lemma 1.

Lemma 1: Three statements, s_α, s_β and s_γ, form an atomicity violation $(s_\alpha, s_\beta, s_\gamma)$ at runtime $\Leftrightarrow$ $\exists$ statement $s_\theta \in between(s_\alpha, s_\beta)$, s_θ is executed at the same time as s_γ.

Here, $between(s_\alpha, s_\beta)$ returns a set, denoted as the bset (short for between set), of statements that may be executed between s_α and s_β locally by the same thread. One way to compute it is to collect the statements on every path from s_α to s_β, which can be implemented efficiently using the standard forward data-flow analysis[2].

Mapping to the Petri net, Lemma 1 is reiterated as Lemma 2.

Lemma 2: Three statements, s_α, s_β and s_γ, form an atomicity violation $(s_\alpha, s_\beta, s_\gamma)$ at runtime. $\Leftrightarrow$ in the Petri net, where p_α, p_β and p_γ model the statements s_α, s_β, and s_γ, respectively, $\exists$ place $p \in between(p_\alpha, p_\beta)$, p is reached at the same time as p_γ by the execution flow of two threads, or in the Petri net language, two tokens are simultaneously in p and p_γ.

To succinctly describe the tokens in the places, we introduce the marking vector $\mathbf{u}$, where each entry, $\mathbf{u}(p)$, represents the number of tokens of a place, p.

The bset of two places can be computed similar to the bset of two statements. It does not include the transitions or the lock places which do not model "statements". In Figure 2, the bset of places p_3 and p_4 is $\{p_3, p_4\}$. Using the bset, the runtime occurrence of the violation is equivalently described on the Petri net as: $\exists\ p \in \{p_3, p_4\}$, $\mathbf{u}(p) + \mathbf{u}(p_7) = 2$. The sum never exceeds 2 because each place contains at most one token representing the execution flow of its own thread. Remember that we make clones for the same statements under different thread contexts

The counterposition of Lemma 2 also holds, as shown in Lemma 3.

Lemma 3: $\forall$ place $p \in between(p_\alpha, p_\beta)$, $\mathbf{u}(p)+ \mathbf{u}(p_\gamma) \leq 1 \Leftrightarrow$ the atomicity violation does not occur at runtime.

[2]Interested readers may refer to our technique report for the implementation details. http://www.cse.ust.hk/prism/axis/TR.pdf

Lemma 3 naturally provides the solution for constructing the control constraints. For our running example in Figure 2, the control constraints are shown in Equation 1.

$$\begin{cases} \mathbf{u}(p_3) + \mathbf{u}(p_7) \leq 1 \\ \mathbf{u}(p_4) + \mathbf{u}(p_7) \leq 1 \end{cases} \tag{1}$$

As p_3 and p_4 are places of the same thread, at any time, the token representing the execution flow of the thread can at most stay in one of them. Therefore, we have an intra-thread constraint: $\mathbf{u}(p_3)+\mathbf{u}(p_4) \leq 1$. Combining the constraints in Equation 1 and the intra-thread constraint, it is not hard to obtain an equivalent but more succinct constraint, $\mathbf{u}(p_7) + \mathbf{u}(p_3) + \mathbf{u}(p_4) \leq 1$.

We formally generalize the succinct representation of the constraint in Lemma 4.

Lemma 4: Suppose $between(p_\alpha,\ p_\beta)$ consists of places p_α, p_θ, $\dots$, p_β. They form an implicit intra-thread constraint (Equation 2a) as well as the control constraints with a remote place p_γ (Equation 2b to Equation 2e).

$$\mathbf{u}(p_\alpha) + \mathbf{u}(p_\theta) + \cdots + \mathbf{u}(p_\beta) \leq 1 \tag{2a}$$
$$\mathbf{u}(p_\gamma) + \mathbf{u}(p_\alpha) \leq 1 \tag{2b}$$
$$\mathbf{u}(p_\gamma) + \mathbf{u}(p_\theta) \leq 1 \tag{2c}$$
$$\dots \tag{2d}$$
$$\mathbf{u}(p_\gamma) + \mathbf{u}(p_\beta) \leq 1 \tag{2e}$$

then, we have an equivalent but more succinct constraint, i.e., control constraint:

$$\mathbf{u}(p_\gamma) + \mathbf{u}(p_\alpha) + \mathbf{u}(p_\theta) + \cdots + \mathbf{u}(p_\beta) \leq 1. \tag{3}$$

Proof 1: The proposition Equation 3 $\rightarrow$ Equation 2 holds as each place contains non-negative number of tokens.
The proposition Equation 2 $\rightarrow$ Equation 3 also holds. Suppose $between(p_\alpha,\ p_\beta)$ contains k places, then we apply the following linear transformations to get Equation 3. Multiply Equation 2a by (k-1), we get Equation 4.

$$(k-1)(\mathbf{u}(p_\alpha) + \mathbf{u}(p_\theta) + \cdots + \mathbf{u}(p_\beta)) \leq (k-1) \tag{4}$$

Add Equation 4 with Equation 2b, Equation 2c, $\dots$, Equation 2e, we get Equation 5.

$$k(\mathbf{u}(p_\gamma) + \mathbf{u}(p_\alpha) + \mathbf{u}(p_\theta) + \cdots + \mathbf{u}(p_\beta)) \leq 2k-1 \tag{5}$$

Divide Equation 5 by k, we get Equation 6.

$$(\mathbf{u}(p_\gamma) + \mathbf{u}(p_\alpha) + \mathbf{u}(p_\theta) + \cdots + \mathbf{u}(p_\beta)) \leq 2 - 1/k \tag{6}$$

As each place contains integer tokens, we further get Equation 3. ∎

Definition 1 (Danger stripe): Given a violation and its succinct constraint in Equation 3, the places appearing in the equation form a danger stripe for the violation. With the assistance of an indicator vector $\mathbf{l}$ indicating each place's presence or absence in the danger stripe with 1 or 0, Equation 3 can be expressed in a vector form: $\mathbf{lu} \leq 1$ [3].

Redundant Constraints. A constraint is redundant if it is subsumed by other constraints. Taking the constraint in Equation 1 for example, the constraint $\mathbf{u}(p_7) + \mathbf{u}(p_3) \leq 1$ is subsumed by $\mathbf{u}(p_7) + \mathbf{u}(p_3) + \mathbf{u}(p_4) \leq 1$. As the SBPI solver works on independent constraints, we conduct a preprocessing step to remove the redundant constraints. In particular, we remove the constraint if the set of places appearing in the constraint are covered by the set of places in another constraint. Note that such reduction is essentially a more natural and rigorous form of the heuristic-based redundancy removal method used by AFix.

C. Solving the Control Constraints

Given the constraint in the form of $\mathbf{lu} \leq 1$, our solution is to generate an augmentation to the Petri net to force it to avoid violation executions, using the Supervision Based on Place Invariants (SBPI) theory. The solution augments the original Petri net with new places and new arcs to guarantee that this constraint is always satisfied in the augmented net. Before describing the SBPI technique, we first introduce the mathematical model of both the Petri net and its dynamic operations.

Definition 2: A Petri net is a bipartite directed graph, denoted by a tuple $\langle P, T, A, W, \mathbf{u}^0 \rangle$. P is the set of places and T the set of transitions. $A \subseteq (P \times T) \cup (T \times P)$ is the set of arcs. $W : A \times N$ is a weight function which returns a natural number for each arc. $\mathbf{u}^0 : P \times N$ is a function which returns the number of tokens initially hosted in a given place.

The Petri net used in our technique is a special type of Petri net called the *pure Petri net*[4], where tokens cannot directly flow back to the source place (no self-loop). Besides, the weight of each of its arcs is 1, and each place can have at most 1 token. The movement of tokens in Petri net is characterized by a 2-dimensional incidence matrix $\mathbf{D}$. Each cell $\mathbf{D}_{ij}$ is either 0 (not connected), 1 (incoming), or -1(outgoing), representing the direction of the transition t_j with respect to the place p_i.

We state the SBPI as follows.

Theorem 1 (SBPI Theorem): If $\mathbf{lu}^0 \leq b$, then we can guarantee that the constraint $\mathbf{lu} \leq b$ holds for any possible marking $\mathbf{u}$ by adding a new place M. The newly added M is characterized by its initial marking $\mathbf{u}^0_M$ and its row vector $\mathbf{D}_M$ in the incidence matrix $\mathbf{D}$, which should satisfy the following condition:

$$\mathbf{u}^0_M + \mathbf{lu}^0 = b \tag{7}$$

and

$$\mathbf{D}_M + \mathbf{lD} = \mathbf{0} \tag{8}$$

[3]$\mathbf{l}$ is a row vector, $\mathbf{u}$ is a column vector.

[4]A standard Petri net result states that any Petri net can be reduced to pure Petri nets [22].

The added place enforces the maximal permissiveness in the sense that it preserves as many executions as possible while prohibiting only the violation-inducing executions.

We interpret Theorem 1 intuitively as follows. The constraint that the number of tokens in the danger stripe cannot exceed a bound b always holds if, in the augmented Petri net, the total number of tokens initially in both the danger stripe and M (newly added place) is b, defined by Equation 7, and the net change of the number of tokens in the danger stripe and M is always 0, defined by Equation 8. Based on the two equations, we can easily compute the initial marking of the new place as: $b - \mathbf{lu}^0$ and the new arcs, as well as their directions, $\mathbf{D}_M = -\mathbf{lD}$.

Let us apply the SBPI technique to fix the atomicity violation in the running example, of which the incidence matrix is

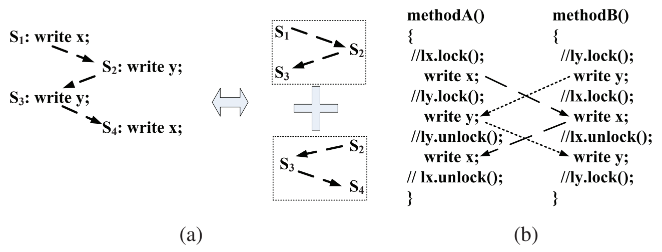
$$\mathbf{D} = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \\ p_9 \\ L \end{array} \begin{array}{c} \begin{array}{ccccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 \end{array} \\ \begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

The initial marking is $\mathbf{u}^0$= $[1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1]^T$ (the places are ordered as p_1, p_2, $\ldots p_9$, L.). The constraint is $\mathbf{u}(p_3) + \mathbf{u}(p_4) + \mathbf{u}(p_7) \leq 1$. In other words, the indicator vector $\mathbf{l}$ is $[0\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 0\ 0]$ and $b = 1$. Applying the SBPI constraint solver, the transition vector of the new place M, $\mathbf{D}_M$, is $-\mathbf{lD} = [0\ -1\ 0\ 1\ 0\ -1\ 1]$ with the initial marking $\mathbf{u}^0_M$ as $b - \mathbf{lu}^0 = 1$. The new place, M, and its connecting arcs, as indicated by $\mathbf{D}_M$, is shown in Figure 2(c).

As shown in Figure 2(c), the new place, M, has two outgoing arcs to t_2 and t_6, and two incoming arcs from t_4 and t_7. The behavior of place M is identical to a lock, as explained in Section II. The arc connecting place M to transition t_2 essentially says, a lock operation should be added to the transition t_2, i.e., the control flow edge represented by t_2 in the CFG. We show how to achieve this in the program code in Section IV.

D. Multi-variable Atomicity Violation

So far our discussion is focused on atomicity violations involving a single shared variable. Our approach can be naturally applied to fix multi-variable atomicity violations, or the so-called atomic-set (ASET) violations [30]. Vizari et al. [30] exhaustively enumerate 11 thread interleaving patterns as the equivalent conditions of ASET violations. For example, the pattern $W^1_x W^2_y W^1_y W^2_x$ describes a violating interleaving, where the first and the third write belong to



```
methodA()             methodB()
{                     {
 //lx.lock();          //ly.lock();
   write x;              write y;
 //ly.lock();          //lx.lock();
   write y;              write x;
 //ly.unlock();        //lx.unlock();
   write x;              write y;
// lx.unlock();        //ly.lock();
}                     }
```

Figure 3: (a) Multiple-variable atomicity violation, the dotted lines depict the forming interleavings of the violation. (b) The deadlocks may be introduced.

a unit of work in one thread and the second and the fourth belong to the other. As shown in Figure 3(a), the interleaving is equivalent to the co-occurrence of the interleavings shown in the two squares. Following the process described above, we can construct two constraints, $\mathbf{u}(s1)+\mathbf{u}(s2)+\mathbf{u}(s3) \leq 1$ and $\mathbf{u}(s2)+\mathbf{u}(s3)+\mathbf{u}(s4) \leq 1$ and add both of them to the constraint solver. Our technique treats ASET violations in a uniform manner with the same guarantees. Comparatively, it is non-trivial to adapt AFix [18] to fix ASET violations because the approach is designed specifically for single-variable atomicity violations.

E. Deadlock-Free Guarantee

Our approach fixes the violations one by one to achieve the fine-granularity concurrency control for each bug. However, due to the interplay between the overlapping bugs, deadlocks may be introduced. For example in Figure 3(b), there are two bugs, each involving variables x and y, respectively. The computed controllers lx and ly (appearing in the comments) may form the cyclic lock acquisition orders, i.e., deadlocks. The work of Wang et al. [34] is used as a postprocessing step to fix deadlocks. We extend the deadlock avoidance to guarantee that our approach does not introduce new deadlocks to the program. In particular, we only fix those deadlocks introduced by our added locks or, in the Petri net language, we only handle the siphons involving the newly added lock places[5].

F. Guarantee of Lock Placement

Our approach guarantees that the lock placement is free of bad lock practices. In this section, we show the guarantee with respect to both the intra-procedural and inter-procedural constructs.

Lock Placement when Branches or Loops are Present. Our approach generates only the matched lock–unlock operations. It never generates bad lock practices such as double locking, double unlocking, missed locking, and missed unlocking. It is well known [18] that the careless manual fixing easily leads to bad lock practices such as the double locking. Surprisingly, our approach avoids bad lock practices without any special treatments. The key insight is that, the SBPI theory guarantees that the places in a danger stripe together with the added supervisory place always contain an invariant number of tokens in total[6]. In the case that the bad lock practices are present, e.g., missed unlocking, the execution along a certain path may cause inconsistencies in the total tokens in both the danger stripe and the supervisory place. The formal proof and a concrete example is shown in our technique report[7].

Lock Placement when s_α and s_β are in Different Functions. (Remember that s_α, s_β, s_γ form a violation.) When s_α and s_β are in different functions, the SBPI solver may also place the lock acquisition and the lock release in different functions. Such a placement can cause errors if, for example, the method containing the lock release gets invoked under a different calling context where no locks are acquired. To prevent such errors, we place a restriction on the statements s_α and s_β that they must be enclosed in the same method. In the case that they are not in the same method, we preprocess the violations to make sure s_α and s_β are mapped to the call sites in their lowest common ancestor (LCA) method.

G. Discussion of Limitations

A major threat to the validity of our high performance claim is that we use the degree of concurrency as the dominant factor for the performance measurement. We minimize the number of locking operations only under the condition that the maximal concurrency degree is achieved. In fact, researchers ([11], [15]) often consider the number of locking operations as an equally important factor of the performance. The concurrency degree may be sacrificed sometimes for reducing the number of locking operations. Our technique has the consequence that, the locks we introduce may be in the hot loops, leading to many the locking operations. We downplay the importance of locking frequency for two reasons. First, due to the JVM optimization [4][8], the uncontended locking operations are extremely fast, and their cost is negligible if no hot loops are present. Second, our empirical experience shows that atomicity violations rarely happen in the hot loops perhaps because the high frequency of execution can easily expose the bugs. Nevertheless, as a mitigation measure, we conduct the cycle detection to inform the developers that some locks are placed in loops or recursions.

Our input violation bugs are detected by Pecan [16], which may contain benign violations that, according to Chew et al. [7], do not affect the correctness of the program

[5]More details and formal proof are in our technique report.http://www.cse.ust.hk/prism/axis/TR.pdf

[6]Actually, this is the reason why the technique is called SBPI, and the reason why the constraint can be satisfied.

[7]http://www.cse.ust.hk/prism/axis/TR.pdf

[8]http://blogs.oracle.com/dagastine/

or even required in some cases. Since Pecan does not classify benign violations, fixing them may incur unnecessary overhead. However, we think this is related to the quality of the bug reports and does not undermine our contribution.

IV. Implementation

We implement our approach as a tool, Axis[9], on top of the Soot framework. Axis contains four components: the Petri net builder, the constraint constructor, the constraint solver, and the patcher.

The construction of Petri nets is based on the depth-first traversal of the call graph and the intra-procedural control flow graphs. We construct Petri nets by substituting each node-edge-node structure with the place-arc-transition-arc-place structure. For methods invoked under different contexts, we clone the Petri nets of these methods to get context sensitivity. If threads are created in recursions or loops, we further clone the Petri subnet rooted at the `run` method. The constraint construction is built on top of the forward data flow analysis in Soot.

Our patcher works directly on the bytecode instead of the source code. The implementation of the patcher is full of challenges. We highlight important ones as follows.

Patching Locks. The instrumentator adds the locking operations, i.e., the monitorenter and monitorexit instructions, to the program according to the result of the solver. The patched lock is created as a static variable of the class that contains the program entry point (`main` method). Since this class is usually loaded first, we ensure the lock variables are always properly initialized before being used.

Exception Handling. Exceptions may be thrown between a pair of added locking operations, which make the execution skip the monitorexit instruction. To make sure that our patcher works in such case, we add an exception handler directly after the monitorexit, which captures the exceptions thrown between the monitorenter and the monitorexit statements and also from our injected exception handlers.

Control Flow Interception. As shown in Section III-C, our solution is in the form of places connecting to the transitions. From the view of CFG, we need to inject the lock or unlock operations to the control flow edges modeled by the transitions. However, the control flow edges are not explicitly represented in the code. Given a control flow edge E (from m to n), where we want to inject the unlocking operation, naively injecting monitorexit just before the target statement n is problematic because other edges with n as the target are also affected. We explicitly represent the control flow edges in the code and isolate the effect of monitorexit to the control flow edges as follows. Given the edge E, we change it to $m \overset{jumpto}{\rightarrow} nop \overset{jumpto}{\rightarrow} n$. In this way, the nop, having a single parent and single child, explicitly represents the flow edge. Then, injecting the monitorexit to just before or after nop (but not across the jumping statement) will not affect other control flow edges.

[9]We make it publicly accessible: http://www.cse.ust.hk/prism/axis

V. Evaluation

In this section, we aim at answering the following research questions.

1) Does our approach automatically fix the atomicity violations?
2) How does our approach affect the concurrency degree of the original program as compared to the state of the art approach?
3) Does our approach introduce new deadlocks to the original program?

To answer these questions, we implemented our approach as a tool, Axis, and used it to fix the atomicity violations found in a set of popularly used subjects ([16], [25]), as listed in Table I. The subjects contain three large scale programs, Openjms, Derby, and Jigsaw, which are, respectively, the open source implementation of JMS specification[10], Apache's database management system, and W3C's web server platform. Table I shows, in column two and three, the number of application methods ($\#Method$) and the number of statements ($\#Stmt$)[11]. We collect the bugs using the violation detection tool, Pecan. All studies are performed on a x86_64 Dell workstation with 3.0GHz quad-core Intel Xeon X5450 processors based on Core 2 micro-architecture (8 cores total). The server has 16GB RAM and 6M L2 caches, runs Ubuntu 8.04 with a Linux 2.6.22 kernel, and uses Sun's 64-Bit 1.6.0 JVM. For each program, we run 20 runs to collect the data.

The first question can be answered easily by measuring the effectiveness of Axis. We run the violation detection tool, Pecan, upon the patched versions of the subjects with the same program input. Pecan detected no further violations for any of the 13 subjects. We report our investigation of the remaining two questions as follows.

A. The Study of the Bug Fixing Process

In this section, we study various important characteristics of the bug fixing process of Axis itself. In Table I, we first report, for each subject, the space and time usage of Axis in terms of the number of bugs fixed ($\#bug$), the size of the constructed Petri net ($size$), the time of fixing the violations (T_{av}), and the time of fixing the deadlocks (T_{dl}). We show the time of the call graph construction (T_{cg}) as the reference.

The data show that the constructed Petri nets, despite being context-sensitive, contain a small number of nodes compared to the total number of program statements. The largest number of nodes in our evaluation (Jigsaw) is around 25K, which can easily fit into the memory of a desktop machine. The time for fixing the violations is almost negligible,

[10]http://java.net/projects/jms-spec/pages/Home

[11]We conduct the analysis on the Soot's jimple IR. Hereafter, the statement means the Jimple IR.

```
 postAttributeChangeEvent() {
1      ...
2      if (( attrListener != null ) ) {
3         AttributeChangedEvent evt =
4             new   AttributeChangedEvent(getResource(),
5                                     attributes[idx],
6                                     newvalue);
7         fireAttributeChangeEvent(evt);
8      }
9  }
```

(a)

```
postAttributeChangeEvent() {
      ...
      l1.lock();
      if (( attrListener != null ) ) {
            l2.lock();
         AttributeChangedEvent evt =
             new   AttributeChangedEvent(getResource(),
                                     attributes[idx],
                                     newvalue);
           l2.unlock();
        fireAttributeChangeEvent(evt);
      }
      l1.unlock();
  }
```

(b)

```
postAttributeChangeEvent() {
      ...
      l.lock();
      if (( attrListener != null ) ) {
         AttributeChangedEvent evt =
             new   AttributeChangedEvent(getResource(),
                                     attributes[idx],
                                     newvalue);
        fireAttributeChangeEvent(evt);
      }
       l.unlock();
  }
```

(c)

Figure 4: The fixing of the violations in Jigsaw. (a) The violation. (b) The fixing of our approach. (c) The fixing of AFix approach.

less than 1 second in all subjects. The deadlock avoidance analysis also scales well with respect to the program size, taking 30 seconds in the worst case.

The two columns under the category of Locks reports the number of locks introduced by Axis and AFix, respectively. For the three large subjects, to fix the same number of bugs, AFix on average generates 70% fewer locks compared to Axis. This is because, its heuristics-based merging step cannot precisely reason about the interplay between the overlapping bugs and conservatively protects several unrelated bugs with the same lock. This directly translates to the coarse-granularity concurrency control and the longer blocking time on the locks. We use a real example in the subject Jigsaw to further clarify this point.

Figure 4(a) is a simplified code snippet from Jigsaw that shows the interplay between two atomicity violations. In the first bug, local statements line 2 and line 7 accessing the variable $attrListerner$, are interleaved by a remote statement (not shown), s_γ, that also accesses this variable. In the second bug, line 5 is the remote statement, accessing the variable $attributes$, that interleaves with a pair of local statements (not shown), s'_α and s'_β, that access the same variable. Note that the two local statements in the second bug can be executed in parallel with the statement, line 7, in the first bug because they modify unrelated program states. As shown in Figure 4(c), AFix merges the two violations as one and protects them with a common lock. As the result, line 7, a long computation, and the sequence between s'_α and s'_β cannot be executed concurrently. Our fix, as shown in Figure 4(b), generates two locks, one for each bug. Line 7 can be allowed to execute because s'_α and s'_β now acquire a different lock. Our approach, without any fear of deadlocks, makes use of nested locking to reduce the lock contention.

B. Performance of the Patched Code

In this study, we assess the performance impact of the generated patches. For each subject, we compare the performance of four versions, the original version (Orig), the version generated by AFix, the patched versions by our technique with Axis (Axis-DA) and Axis with the deadlock avoidance disabled (Axis-noDA). Since the original AFix tool was developed for C programs, we have reimplemented its algorithm faithfully and released our implementation for further scrutiny[12]. Our performance study consists of two experiments: the first studies the computational throughput of patched versions compared to the original, the second studies the scalability of the throughput with an increasing number of threads.

In the performance category of Table I, we show the performance measurements of the subjects with the number of threads fixed at eight, which is the number of cores on our test platform. The first seven subjects use few synchronization operations and have no more than three violations to fix. For this reason, the patches generated by Axis-noDA and AFix are almost identical. The performance of each version is, therefore, more or less the same with minor fluctuations due to the thread scheduling. For subjects Cache4j, Hedc and Specjbb, The Axis-DA version, however, introduces a higher overhead of 20%, 5% and 4%. This overhead is the cost of deadlock-freeness through the use of additional locks.

The result for the three large subjects shows the superiority of Axis. For the subject OpenJMS, the patched version with no deadlock avoidance (Axis-noDA performs almost as well as the original version, which contains violations. And, it outperforms the AFix version by 8%. With the deadlock-free guarantee, our Axis-DA version is 4% slower than the AFix version, which does not provide a deadlock-free guarantee. For the subject Jigsaw, we are unable to obtain the stable throughput data for Axis-noDA and AFix

[12]AFix Java implementation. URL: http://www.cse.ust.hk/prism/axis

Table I: The metrics of the fixing process and the performance of the patched programs. In the performance column, we use millisecond (ms) as the unit. One exceptional case is Specjbb, where we use its own specific unit to measure the performance, bops (business operations per second). Higher performance corresponds to higher bops value.

	Program properties		Metrics of the fixing					Locks		Performance			
Program	$\#Method$	$\#Stmt$	$\#bug$	$size$	$T_{cg}(sec)$	$T_{av}(ms)$	$T_{dl}(ms)$	Axis	AFix	$Orig$	$Axis\text{-}noDA$	$Axis\text{-}DA$	$AFix$
MergeSort	59	695	1	494	49.6	9	29	1	1	20	21	20	20
StringBuffer	39	361	1	148	51.2	9	16	1	1	10	10	11	11
ArrayList	24	221	2	328	31.7	3	24	2	2	13	15	15	14
LinkedList	43	347	2	336	31.9	3	32	2	2	10	11	13	11
HashSet	10	78	3	264	31.3	2	11	2	2	23	28	28	29
TreeSet	17	115	2	432	33.4	2	24	1	1	25	27	27	25
RayTracer	59	695	2	378	33.2	18	22	2	1	17	19	20	21
Cache4j	197	2948	2	352	50	121	362	2	2	35	36	42	37
Hedc	136	1187	3	386	59	113	384	2	2	5441	5513	5720	5533
SpecJBB	757	21244	31	450	61	134	371	2	2	80337	78078	75263	77970
OpenJMS	19476	302004	296	17972	128	574	1835	87	28	5415	5501	6193	5936
Jigsaw	11227	222118	754	25570	65.5	577	32492	50	10	960	-	1135	-
Derby	20964	321142	330	16529	79.4	603	24034	74	13	503	520	557	548

because the patched versions easily lead to deadlock with eight threads running. This case shows the importance of the deadlock avoidance in the patch. For the subject Derby, the Axis-noDA version is 6% faster than the AFix version. The Axis-DA version, with deadlock-free guarantee, incurs only 2% higher overhead compared to the AFix version. In general, the overhead incurred by the Axis-noDA version is around 40% of the overhead of the AFix version.

In Figure 5, we show a study of the scalability of the patched code by both our technique and AFix with respect to the increasing number of threads, using the large scale subjects OpenJMS, JigSaw and Derby. For the subject OpenJMS, when the thread number is larger than 4, the Axis-noDA version outperforms the AFix version by around 9%. The Axis-DA version incurs 3% (2 threads) to 20% (12 threads) overhead. Compared to the AFix version, it is 1% (4 threads) to 6% (12 threads) slower, but it guarantees the deadlock-freeness. For the subject Jigsaw, both the Axis-noDA and AFix versions easily lead to deadlocks. The Axis-DA version incurs a slowdown of 9% (4 threads) to 24% (2 threads). For the subject Derby, the Axis-noDA version outperforms the AFix version by around 6% when the thread number increases to 8. The Axis-DA version, incurring 6% (2 threads) to 11% (12 threads) overhead, is slightly slower than the AFix version by 2% on average. As seen, our technique exhibits a better performance than AFix as the thread number increases. An explanation for the observation is our approach leads to fewer contentions of the same locks and allows higher concurrency degree.

One interesting observation from Figure 5 is that, when the thread number is 1, the performance of all versions is similar. Such an observation illustrates that the added locking operations are often not in the hot loops, and introduce negligible overhead, if we do not consider blocking time, as compared to the original version. This confirms with our assumption that the locking frequency is not a key factor that influences the performance of our subjects.

To conclude, our Axis-noDA outperforms AFix approach by 6% to 9% in the realistic applications. For conservatively guaranteeing deadlock-freeness, our Axis version often incurs a moderate (average 10%) overhead.

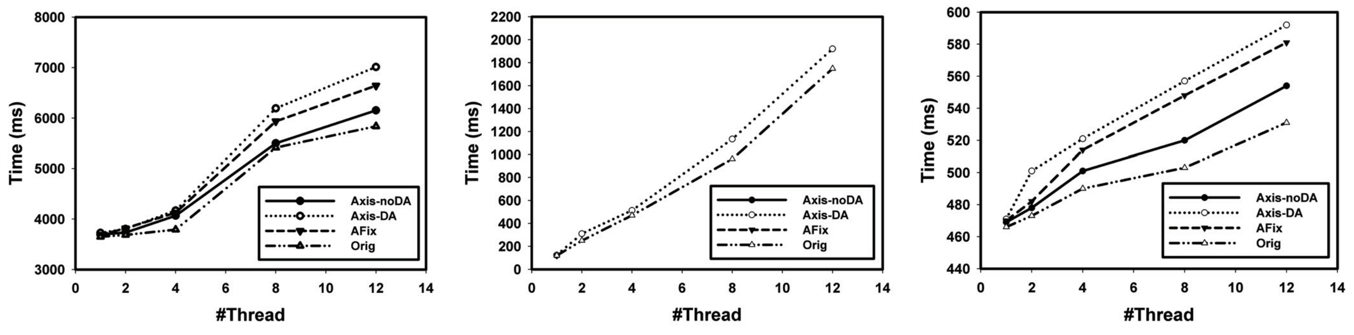
C. Safety of the Generated Code

In this study, we aim to assess if our patches introduce deadlocks in the subjects. We could use existing deadlock detecting techniques to quantify this property. However, we found that simply by sampling the runs for a fixed number of times is already sufficient to reveal interesting insights. We run each patched program 20 times and observe whether deadlocks happen without any interference to the scheduling mechanism.

As shown in Table II, the deadlocks can easily happen with even 20 uncontrolled runs. For Openjms, each of the version with the Axis-noDA patch and the version with the AFix patch gets 2 deadlocks when the thread number reaches 8. When the thread number reaches 12, more deadlocks (7 and 5) incur in these two versions. For Jigsaw, the two patched versions of Jigsaw always get deadlocks at runtime when there are multiple threads. For Derby, when the thread number reaches 12, the patched versions by Axis-noDA and AFix incur 11 deadlocks and 7 deadlocks, respectively. Comparatively, we observe no deadlocks for the patched version by Axis-DA. This observation shows the deadlock is a serious problem to treat while fixing of violations. It is worthwhile to pay a higher cost in the Axis-DA patches to avoid them.

VI. Related Work

Lock allocation ([11], [15], [21]) is akin to the automatic fixing of violations. Its goal is to infer the synchronization constructs to guarantee the specified atomicity properties



(a) (b) (c)

Figure 5: (a) Openjms. (b) Jigsaw. (c) Derby.

Table II: The fixing process of Axis-noDA and AFix introduce deadlocks. For a subject, we show the deadlocks for both its Axis-noDA version and AFix version.

Patched Program		$T = 2$	$T = 4$	$T = 8$	$T = 12$
Openjms	Axis-noDA	0	0	2	7
	AFix	0	0	2	5
Jigsaw	Axis-noDA	20	20	20	20
	AFix	20	20	20	20
Derby	Axis-noDA	0	0	0	11
	AFix	0	0	0	7

and the high performance. Recent lock allocation work ([14], [29]) makes use of the shape properties to generate the fine-granularity locks, so that multiple operations can simultaneously access different parts of a data structure.

Automatically inferring the atomicity property, i.e., the atomic region, is orthogonal to our approach. Vechev et al. [31] propose an assertion-directed approach to iteratively enlarge the atomic region until the assertion is satisfied. Besides, trace analysis ([23], [35]) can also be used to extract the atomicity property from the correct runs.

Self-healing approach [5], [7] detects the faults or errors at runtime and generates dynamic patches so that the executions recover from the anomalies. However, it lacks the comprehensive off-line analysis required to generate high-quality patches. Lucia et al. [20] propose a specific architectural support for dynamically avoiding atomicity violations.

Deterministic execution system [3] or deterministic language support [2] guarantees deterministic behaviors of the program by construction. They still face many challenges such as high runtime overhead or extra manual annotations. An alternative approach, checking the determinism through the testing [24], complements our approach as it can be used to filter out the benign violations.

Petri net is widely applied to various software engineering tasks, e.g., analyzing the performance of software system [9], modeling the protocol of software systems [37], modeling the concurrency programs ([12], [34]).

VII. Conclusions

We have presented an automatic approach to fix atomicity violations through solving the SBPI control constraints, using the SBPI control theory as the theoretical foundation. Given the bug reports of dynamically detected violations, our technique models the offending program statements and their calling contexts using Petri nets. The violations themselves are transformed to a set of linear inequalities with respect to the Petri net structures, which we call control constraints. We then use the SBPI constraint solver to generate an augmented Petri net that: 1. maximally respects the degree of concurrency in the original design; 2. guarantees not to introduce new deadlocks. We implemented our technique as a tool, Axis, and evaluated it against a set of popular concurrent programs. The evaluation shows Axis is both more scalable and effective when compared to the related work.

Acknowledgments

We first thank Yin Wang and Terence Kelly from HP Labs for their patient guidance throughout this work. We also thank the anonymous reviewers for their constructive feedback. This research is supported by RGC GRF grant 622208 and 622909 and by the HKUST RFID Research Center.

References

[1] A time to patch ii: Mozilla. http://voices.washingtonpost.com/securityfix/2006/02/a_time_to_patch_ii_mozilla.html.

[2] Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA*, 2009.

[3] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[4] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for java. In *PLDI*, 1998.

[5] Luciano Baresi and Sam Guinea. Self-supervising bpel processes. *IEEE Trans. Software Eng.*, 37(2), 2011.

[6] Reimer Behrends, R. E. K. Stirewalt, and Laura K. Dillon. Avoiding serialization vulnerabilities through the use of synchronization contracts. In *Austrian Computer Society*, 2004.

[7] Lee Chew and David Lie. Kivati: fast detection and prevention of atomicity violations. In *EuroSys*, 2010.

[8] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *OOPSLA*, OOPSLA '99, 1999.

[9] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. *Model-Based Software Performance Analysis*. Springer, 2011.

[10] Crispin Cowan, Heather Hinton, Calton Pu, and Jonathan Walpole. The cracker patch choice: An analysis of post hoc security techniques1. *Information Systems Security*, 2000.

[11] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL*, 2007.

[12] Azadeh Farzan and P. Madhusudan. Causal atomicity. In *CAV*, 2006.

[13] Cormac Flanagan and Stephen N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.

[14] G. Golan-Gueta, N. Bronson, A. Aiken, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic fine-grain locking using shape properties. In *OOPSLA*, 2011.

[15] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *PACT*, 2007.

[16] Jeff Huang and Charles Zhang. Persuasive prediction of concurrency access anomalies. In *ISSTA*, 2011.

[17] M. V. Iordache and P. J. Antsaklis. Supervision based on place invariants: A survey. *Discrete Event Dynamic Systems*, 16:451–492, December 2006.

[18] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[19] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.

[20] Brandon Lucia, Joseph Devietti, Luis Ceze, and Karin Strauss. Atom-aid: Detecting and surviving atomicity violations. *IEEE Micro*, 29, 2009.

[21] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, 2006.

[22] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, apr 1989.

[23] Abdullah Muzahid, Norimasa Otsuki, and Josep Torrellas. Atomtracker: A comprehensive approach to atomic region inference and violation detection. In *MICRO*, 2010.

[24] Adrian Nistor, Darko Marinov, and Josep Torrellas. Instantcheck: Checking the determinism of parallel programs using on-the-fly incremental hashing. In *MICRO*, 2010.

[25] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, 2008.

[26] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[27] Eric Rescorla. Security holes... Who cares? In *SSYM'03: Proceedings of the 12th Conference on USENIX Security Symposium*, pages 75–90, Berkeley, CA, USA, 2003. USENIX Association.

[28] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. Penelope: weaving threads to expose atomicity violations. In *FSE*, 2010.

[29] Gautam Upadhyaya, Samuel P. Midkiff, and Vijay S. Pai. Using data structure knowledge for efficient lock generation and strong atomicity. In *PPoPP*, 2010.

[30] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[31] Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

[32] Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. Model checking programs. In *ASE*, 2000.

[33] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, 2008.

[34] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott Mahlke. The theory of deadlock avoidance via discrete control. In *POPL*, 2009.

[35] D. Weeratunge, X. Zhang, and S. Jagannathan. Accentuating the positive: Atomicity inference and enforcement using correct executions. In *OOPSLA*, 2011.

[36] Charles Zhang. Flexsync: An aspect-oriented approach to java synchronization. In *ICSE*, 2009.

[37] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE*, 2006.