Finding Incorrect Compositions of Atomicity

Peng Liu* lpxz@ust.hk
Julian Dolby† dolby@us.ibm.com
Charles Zhang* charlesz@cse.ust.hk

*Hong Kong University of Science and Technology China
†IBM Thomas J. Watson Research Center United States

ABSTRACT

In object-oriented code, atomicity is ideally isolated in a library which encapsulates shared program state and provides atomic APIs for access. The library provides a convenient way for programmers to reason about the needed synchronization. However, as the library exports a limited set of APIs, it cannot satisfy every unplanned atomicity demand; therefore, clients may have to compose invocations of the library APIs to obtain new atomic functionality. This process is error-prone due to the complexity of reasoning required, hence tool support for uncovering incorrect compositions (i.e., atomic compositions that are implemented incorrectly) would be very helpful. A key difficulty is how to determine the intended atomic compositions, which are rarely documented. Existing inference techniques cannot be used to infer the atomic compositions because they cannot recognize the library and the client, which requires understanding the related program state. Even if extended to support the library/client, they lead to many false positives or false negatives because they miss the key program logic which reflects programmers' coding paradigms for atomic compositions.

We define a new inference technique which identifies intended atomic compositions using two key symptoms based on program dependence. We then check dynamically whether these atomic compositions are implemented incorrectly as non-atomic. Evaluation on thirteen applications shows that our approach finds around 50 previously unknown incorrect compositions. Further study on Tomcat shows that almost half (5 out of 12) of discovered incorrect compositions are confirmed as bugs by the developers. Given that Tomcat is heavily used in 250,000 sites including Linkedin.com and Ebay.com, we believe finding multiple new bugs in it automatically with relatively few false positives supports our heuristics for determining intended atomicity.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2491435

General Terms

Reliability, Experimentation, Measurement

Keywords

Atomic compositions, concurrent programming, program dependence, static analysis, predictive analysis

1. INTRODUCTION

The difficulty of debugging concurrent programs has inspired the development of a range of tools, based on concepts from race conditions to higher-level concepts like atomicity [8]. Atomicity expresses the intuitive idea that shared state must be accessed by some code without interference. Ideally, atomicity is isolated in a library which encapsulates the shared state and provides the atomic APIs for accessing it without interference. Consider the code snippet (Listing 1) from the Bayesian application [20]: `learnerNet` is an instance of the class `BayesianNet`. The class `BayesianNet` and the referenced class `Node` (not shown) form a library, which encapsulates the fields defining the structure of the Bayesian net. The library also provides two atomic APIs, `hasEdge`, which checks atomically the existence of an edge between two nodes `fromId` and `toId`, and `applyOp`, which inserts atomically an edge between the nodes.

Listing 1. Code snippet of Bayesian application

```
1 isTaskValid= true;
  if(op==INSERT)
3 {
    if(learnerNet.hasEdge(fromId, toId))
5       isTaskValid=false;
  }
7 else{...}
  if(isTaskValid)
9   learnerNet.applyOp(op, fromId, toId);
```

Many promising approaches [31, 28, 27] have been proposed recently to test whether the library implements its own atomic APIs correctly, which have been relatively successful for two reasons: (1) the code responsible for the atomic accesses can be easily identified as the library API methods; (2) the library typically involves a small code base which can be tested exhaustively. However, even if the library implements its atomic APIs correctly, the application may still malfunction because the client code composes new functionality with the library APIs that is intended to be atomic, but does so incorrectly. For example, in Listing 1, two invocations (line 4 and line 9) of library APIs are composed at the client side

to realize new atomic functionality, i.e., to add a given edge between two nodes only if they are not connected by any existing edge.

In general, client-side atomic composition commonly exists because the library designers cannot predict all desired atomic usages. Unfortunately, client-side atomic compositions may be implemented incorrectly, due to the inherent difficulties in reasoning about concurrent behavior: (1) clients compose quite often, e.g., 100+ compositions per evaluated large program (Column *AC* in Table 1); (2) the reasoning for each composition is non-modular, e.g., each studied buggy composition spans multiple methods. Incorrect compositions, i.e., atomic compositions that are implemented incorrectly, commonly lead to unwanted behaviors. For example, in Listing 1, if the two invocations are interleaved non-atomically by an invocation in another thread that inserts an edge between the two nodes, the connectivity state returned by the former invocation becomes stale, the latter invocation which depends on the stale state proceeds to insert another edge. Consequently, two edges are inserted between the two nodes, violating the invariant.

We identify the problem of the incorrect client-side composition and address it automatically in this paper. Based on the automatic discovery of the library and the client, we infer the intended client-side atomic composition automatically, after which we adopt existing atomicity violation detection analyses [8, 26] to find the incorrect implementation of the atomic composition.

Automatic inference of atomic composition is required to free programmers from the daunting task of manually specifying 100+ compositions per large program. However, the inference is challenging. The first challenge lies in recognition of the library and the client, which is a prerequisite for atomic composition inference. Existing inference [18, 22, 40, 4, 17, 34] of atomicity, which reasons about the field accesses directly, cannot recognize the library or the client. Specifically, recognition of the library requires understanding what fields are related, e.g., some fields from the class `BayesianNet` and the class `Node` are related in describing the structure of the Bayesian net, therefore, both classes are included in the same library. Most existing inference techniques [18, 22, 40, 4, 34], which understand only the single-variable atomicity, would isolate the fields in different libraries and accordingly miss the atomicity on the related fields. Furthermore, recognition of the irrelevant client-side uses of a library requires understanding independent portions of the library state, e.g., the independence between the fields describing the structure of the net and the fields describing the domain-specific values stored in the net. The second challenge posed to the inference is it needs to achieve high fidelity, i.e., it neither misses many real atomic compositions nor introduces many false ones. Existing inference [18, 22, 4, 17, 34] of atomicity, even if extended to support the library/client, fails to achieve the high fidelity. Specifically, the inference techniques [18, 22, 4, 34] identify the atomicity with the dynamic instruction adjacency or the lexical adjacency, which introduce too many atomic compositions conservatively; the inference techniques [17] identify the atomicity with the frequency-based (or statistic-based) symptom, which miss many real atomic compositions. For example, the atomic composition in our running example is missed because it appears only once.

We propose a new approach to address these challenges. First, we identify the library and the client based on the notion of *atomic set* [36, 5], which characterizes a group of related fields. As the related fields are linked by field references and occur in existing atomic regions, we infer them from existing atomic regions and the field reference logic indicating the relevance. Second, to achieve the high fidelity, we identify atomic compositions by finding definitive code paradigms for them. According to our study, most compositions coded as atomic fall into two general cases:

- One invocation leaks the state of the atomic set encapsulated by the library and the other invocation uses it, e.g., the latter invocation in the exemplary atomic composition (Listing 1) uses the connectivity state leaked by the former invocation. Atomicity is required because otherwise the latter invocation would use the stale state and lead to unwanted behavior, e.g., inserting multiple edges between two nodes in our running example.

- Instead of depending on the return value of each other, two invocations are inseparable and complement each other to fulfil a combined atomic functionality, e.g., the invocations `setCity` and `setZipcode` are combined to achieve the atomic functionality `setCityAndZipcode`.

Both coding paradigms can be captured by the program dependence logic, and we use them as definitive symptoms to identify the atomic compositions.

Once we have identified the compositions intended to be atomic, existing atomicity violation detection [26, 13] can check whether the compositions are actually atomic. We customize existing predictive analyses [30, 13, 38] to find incorrectly implemented compositions. Predictive analysis searches for non-atomic interleavings of the atomic composition by adjusting the scheduling order of events. Specially, we customize the analysis so that it is aware of the library APIs involved in the atomic compositions. Since some atomic compositions reported by the conservative static inference may be false, our dynamic analysis does runtime checking to filter out such false positives. Our dynamic analysis is optimized for our specific task: it analyzes at the invocation level and analyzes only invocations accessing the same atomic set. This reduces the number of events to analyze and reduces the interleaving space to explore without compromising effectiveness.

We implement our approach as a tool ICfinder and evaluate it on a set of large scale applications such as Tomcat. ICfinder has been successfully evaluated by the ESEC/FSE artifact evaluation committee and found to meet expectations. Our results show that ICfinder finds around 50 incorrect compositions in all applications. Further manual study of our Tomcat results shows that five out of the twelve incorrect compositions identified in Tomcat are bugs as confirmed by the developers. Considering that Tomcat is heavily used in around 250,000 sites [3] including the famous ones [37] such as Linkedin.com and Ebay.com, and the easy bugs have already been found, we believe finding newly reported bugs automatically supports our heuristics for detecting incorrect compositions. Overall, the static analysis of ICfinder finds up to 391 atomic compositions for a program. More than half of them appear infrequently and are missed by the previous statistic-based approach. The efficient dynamic analysis of ICfinder finishes within 1 second.

We make the following contributions in this paper.

1. We identify the problem of incorrect compositions of atomic library APIs, i.e., the compositions which are intended as atomic but implemented as non-atomic.

2. We evaluate our approach extensively on a set of large applications and conduct the empirical study to confirm the severity of incorrect compositions.

3. We propose an automatic approach to find incorrect compositions. Specifically, we propose high-fidelity inference of intended atomic compositions, which respects atomicity among related fields and respects the inherent program logic involved in atomic composition. We customize existing dynamic analysis to check whether these atomic compositions are implemented correctly.

2. OVERVIEW

Our approach has four steps: first inferring atomic sets of related fields in a given application, next finding libraries that encapsulate them and the clients that use the libraries elsewhere in the program, then inferring atomicity requirements of these uses, and finally attempting to exhibit executions that violate these atomicity requirements.

Inferring Atomic Sets. This step determines groups of related fields that implement abstractions of atomic sets, which require atomicity for keeping state consistent. The atomic set $aSet$ consists of the related fields sharing a single root, each denoted as a reference chain from the root:

$$\{\langle root, f_1.f_2 \dots f_i.f_{i+1} \dots \rangle \,|\, f_{i+1} \in Type(f_i)\}$$

While the atomic set abstraction could be declared by programmers, we infer them automatically. Our static inference is described in Section 3. Besides, the dynamic checking in Section 6 prunes the false positives produced by the static inference.

Identifying Library and Client. Given an atomic set $aSet$, the library consists of the classes encapsulating the fields in $aSet$, which provide the atomic APIs for accessing the fields in $aSet$. The next step is to find the client methods that use the APIs provided by the library to access $aSet$. The details are explained in Section 4. The abstraction of library and client also provides the intuitive bug reports based on the high-level view of APIs, as contrasted with the bug reports based on the low-level view of field accesses.

Inferring Atomic Compositions. The uses of the library in the client methods may or may not form the atomic composition. Learning from existing atomic compositions, we find they exhibit two common symptoms. We formalize the symptoms and use them as the definition for atomic compositions. The formalization and the inference of atomic compositions are in Section 5.

Exhibiting Synchronization Errors. The atomicity requirement of the composition may not be implemented correctly. The final step is the dynamic execution to force the atomicity violations to manifest themselves. The details and optimization of the dynamic analysis are explained in Section 6.

The rest of the paper presents the above steps in details. In addition, Section 7, Section 8 and Section 9 present the implementation, evaluation and related work respectively.

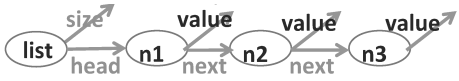

Figure 1. Atomic set in the class `Linkedlist`

3. INFERENCE OF ATOMIC SETS

An Atomic Set [36] denotes a set of heap locations that have some consistency property that is maintained by the *units of work* that operate upon the set. Correctness thus requires that units of work on a given atomic set operate as if they are never interleaved at runtime. Note that a unit of work is subtly different from an atomic region, since it pertains to a specific subset of state.

Definition 1 (Atomic Set)**.** *In object-oriented programs, where objects form reference hierarchies via field references, an atomic set is a set of instance fields, each of which is reachable from the root object along a field chain.*

Figure 1 shows a reference hierarchy rooted at a `LinkedList` object `list`, where ellipses stand for objects and lines stand for field references. The atomic set consists of the highlighted instance fields (grey), `list.size`, `list.head`, `n1.next` and `n2.next`. Each instance field is reachable from the root object `list` along one of the following field chains: `size`, `head`, `head.next` or `head.next.next`.

While atomic set was originally presented as a programming model [5], we need to infer it since we are looking for bugs in existing code. Furthermore, we are looking for incorrect usage of libraries that are themselves well synchronized. Hence, our inference is based on the observation that the related fields in the atomic set are accessed in the same existing synchronized blocks (or methods) and organized by the field reference logic. For each synchronized block (or method), we define an atomic set constituted by fields of the receiver `this` object accessed within the block and also any fields accessed transitively via the receiver's fields. Given all the atomic sets, we merge atomic sets that share fields, to conform with the rule that all atomic sets should be disjoint [5].

Atomicity only applies to shared program state, so we conduct the optimization by leaving out all non-shared program state. We apply the off-the-shelf escape analysis implemented in the Soot [35] compiler framework to prune the thread-local objects.

4. IDENTIFICATION OF CLIENT-SIDE INVOCATIONS

In this section, we identify client-side invocations of library APIs, which access atomic sets encapsulated by the libraries. The next section infers whether such invocations are composed atomically. To identify the invocations, we first determine the library and the client based on the formal description in Definition 2.

Definition 2 (Library Module and Client Module)**.** *A module is a logic unit, which consists of the classes related to a specific property. Two types of modules are of interest. The library module defines the atomic set and the client module uses it. The library module consists of the classes which declare the fields in the atomic set and provide atomic APIs for accessing the fields. The client module consists of the class of which the methods invoke the atomic library APIs. The methods are referred to as client methods accordingly.*

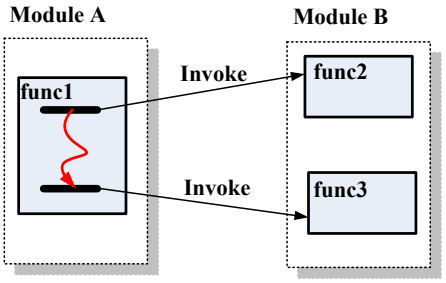

Figure 2. Client module and library module

Figure 2 illustrates the terms: The module B is a library module which provides the atomic APIs, while the module A is a client module, of which the method invokes the APIs in the module B. The modules A and B are often not the same. We also consider the degenerated case where the module A and the module B are the same because the library may compose its own atomic APIs.

We derive the library module directly from each atomic set. Given an atomic set $aSet$, the classes that declare the fields in it form a library module. The client module is identified as the class containing the client method. The client method is further identified as the method containing the invocations of the library APIs that access the atomic set $aSet$.

ALGORITHM 1: The function isClient

```
   Input: The method m, the atomic set aSet
1  foreach stmt: m.stmts() do
2      if stmt.isInvocation() then
3          effects=transitiveSideEffects(stmt);
4          if effects ∩ aSet ≠ ∅ then
5              print: the invocation stmt accesses aSet, m is
                   the client method;
6          end
7      end
8  end
```

We traverse all methods in the application, during which we judge whether each method is a client method following Algorithm 1. If an invocation $stmt$ in the method m accesses a set of instance fields $effects$, according to the side effect analysis (line 3), and some of the instance fields $effects$ belong to the atomic set $aSet$ (line 4), according to the result of the analysis in Section 3, the invocation $stmt$ accesses the atomic set $aSet$.

Specifically, the side effect analysis computes the *transitive* side effects (line 3), i.e., the fields accessed in the transitive callee methods of $stmt$. We compute the side effects by identifying the callee methods in the call graph and resolving the field accesses intra-procedurally. The resultant side effects of each invocation statement are used to judge the accessing relations with respect to each atomic set.

We are trying to infer the atomic sets and clients in a program, and we will get a better approximation given better static analysis. Inevitably, however, static analysis may have unsoundness: it can miss code when it is invoked dynamically using reflection, and it is an over-apporiximation in general of the code. Such issues may reduce the effectiveness of our heuristics, but we show in our results that our static analysis is good enough to give useful results.

5. AUTOMATIC INFERENCE OF ATOMIC COMPOSITIONS

After the invocations potentially accessing the same atomic set are identified, we infer whether they form atomic compositions, using the symptoms capturing the definitive program logic for atomic compositions. According to our observations, compositions of invocations are commonly programmed as atomic if (1) one invocation uses the result produced by the other, or (2) the invocations complement each other indispensably, i.e., in the path conditions that one invocation is executed, the other invocation must be executed. Therefore, we design two symptoms, namely the USE symptom and the complementation symptom, to capture the above scenarios respectively, both based on the program dependence. The USE symptom, described in Property 1, is designed to capture the former atomic composition scenario.

Property 1 (USE symptom)**.** *Given two invocations identified in the client method, if the program dependence exists between them, the invocations should be composed atomically.*

Figure 2 illustrates the atomic composition scenario. The invocations (horizontal lines) in the method `func1` interact with each other via the program dependence (curve): The first invocation leaks the state encapsulated by the library and the second invocation depends on the leaked state. The program dependence link strongly indicates the atomicity of the composition as the interleavings would make the leaked state stale and the program dependence incorrect.

The program dependence is a well studied compiler concept [7, 12, 32], which can be immediate program dependence such as control dependence and data dependence, or the transitive closure of it. Program dependence is often characterized graphically by the program dependence graph [7], where nodes represent statements, edges represent control/data dependences and paths represent program dependences.

On the other hand, the invocations in Figure 2 may not interact with each other but are still composed as atomic. The code from the Specjbb benchmark in Listing 2 illustrates the atomic composition scenario of this kind. Two invocations, which get the `x` scale and `y` scale of an image, should be executed atomically to render a consistent view of the image. As seen, the USE symptom cannot capture such atomic composition as no program dependence exists between the invocations.

Listing 2. Code snippet of Specjbb

```
1 GraphImage.getXscale();
  GraphImage.getYscale();
```

Such atomic compositions are captured by the complementation symptom (Property 2), which specifies that the invocations that are inseparable, i.e., they are executed together or skipped together in any execution, form a combined atomic operation.

Property 2 (Complementation symptom)**.** *Given two invocations identified in the client method, if the invocations dominate and post-dominate each other, the invocations are expected to be composed atomically.*

We check whether the client-side invocations form an atomic composition by checking whether they exhibit either of these symptoms. To check the exhibition of the USE symptom, we examine whether any program dependence exists between the invocations, i.e., whether any path exists between the invocation nodes in the program dependence graph. Checking the path in a graph is a standard reachability analysis. Practical issues about the dependence graph are discussed in Section 7. To check for the complementation symptom, we analyze the dominance and post-dominance relations between the invocations, which is also the standard analysis implemented in open source compilers such as Soot [35] and Wala [33].

6. DYNAMIC CHECKING

After atomic compositions are inferred, we check whether their implementations violate atomicity; we adopt existing atomicity violation detection analyses. We adopt efficient predictive analysis [30, 13, 38], which achieves reasonably good coverage of the buggy interleavings. Predictive analysis aims at deriving, from a normal run, a new buggy run which exhibits non-atomic interleavings. Specifically, the predictive analysis first monitors a normal run to collect the trace and then reorders the events (nodes) in the trace, simulating the rescheduling of the runtime. Finally, the reordered trace is exercised by an execution which enforces the non-atomic interleavings. To serve our specific application, we customize the predictive analysis so that it is aware of the atomic composition. The analysis also prunes atomic compositions falsely reported by the static analysis. Pruning false atomic compositions prevents false alarms, which is important especially for future work that reuses our atomic composition inference. In addition, we optimize the analysis specially for efficiency. The details of the dynamic checking are presented as follows.

Given an atomic composition, our goal is to derive a buggy run (or trace, we may use the terms interchangeably.) from the observed normal run. The buggy run is the run violating atomic-set serializability, a correctness criterion [36] generalized to support the atomic set and commonly adopted. A buggy run can also be defined as the run exhibiting a set of interleaving patterns, which account for all atomic-set serializability violations, according to the study of Vaziri et al. [36]. One exemple bug pattern, $W_u(l)W_{u'}(l)R_u(l)$, states that the write by the unit of work[1] u' to the memory location l interleaves the write and read by the other unit of work u, which leads to the inconsistency between the write and the read in u. To summarize, our goal is to derive the run which exhibits the interleavings specified by the buggy pattern.

Figure 3 illustrates the goal: In the derived run, the remote invocation $I3$ interleaves the atomic sequence from invocation $I1$ to invocation $I2$, or equivalently, the remote invocation $I3$ happens in parallel (denoted by the dotted line) with some event e_{middle} between $I1$ and $I2$ thread-locally. Note that any buggy interleaving can be equivalently expressed by the happen-in-parallel relation [16].

For this example, we exhibit the buggy interleaving by simply reordering $I2$ and $I3$. However, in general, it is non-trivial to exhibit the buggy interleaving, or alternatively,

[1]Here the unit of work corresponds to the execution instance of the atomic code such as the atomic composition or the atomic API.

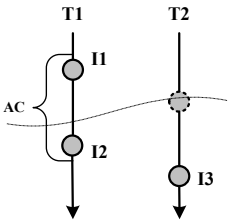

Figure 3. Deriving the buggy run

to enforce the equivalent happen-in-parallel relation. Reordering the events to enforce the happen-in-parallel relation is subjective to many constraints, e.g., the absence of synchronizations and the compatibility of path conditions. Determining whether all the constraints are satisfiable is computationally intractable [24]. Therefore, we approximate the feasibility of the happen-in-parallel relation with a subset of constraints, as described in Lemma 1.

Lemma 1. *Given two events e and e′, they can happen in parallel ⇔ they are not synchronized by the same locks or happened-before relations [15]. Here, e happened-before e′ if and only if one can reach e′ from e in the trace by moving forwards locally or moving along the inter-thread communications (e.g., start, join, wait/notify operations.).*

Given two events from the normal run, if they are not synchronized, we re-schedule one of the events so that they happen in parallel. In practice, we simulate the rescheduling by collecting the trace of the normal run and moving the event forwards/backwards in the trace. After the adjustment, we replay[13] the execution according to the new trace. The replay may fail when the new trace violates the constraints that we do not model. We simply discard such infeasible trace.

Pruning False Atomic Compositions. Due to the conservativeness of the static analysis (Section 3 and Section 4), two invocations accessing disjoint atomic sets may be determined as accessing the same atomic set statically. We prune the false atomic compositions in the monitoring run by checking whether two invocations access the same runtime atomic set. We identify the runtime atomic sets accessed by each invocation as follows. First, we construct the runtime atomic sets by instantiating the root of the static atomic set as runtime instances, which have the same type as the root and are shared among threads. Second, we identify the runtime atomic sets accessed by the invocation based on the dynamic side effect analysis.

Optimization. The optimization is carried out during the reordering of the trace events (nodes). Previous predictive analysis explores the reordering of the memory accesses on the same atomic set. Comparatively, our analysis operates at the invocation level and explores the reordering of the related invocations only, i.e., the invocations accessing the same runtime atomic set. As a result, we reduce the number of events to reorder and reduce the interleaving space to explore. Specifically, we do not explore the interleavings to the internal of the basic library API methods, i.e., the methods which do not compose other library APIs, because they are already well synchronized by the library module.

7. IMPLEMENTATION

Program Dependence Graph. We implement the computation of program dependence in Soot[35]. Soot already provides the implementation for computing the control dependence and the intra-procedural data dependence. We compute the inter-procedural data dependence as follows: (1) we add special data dependence edges to express the inter-procedural argument passing and function return; (2) we use the side effect analysis to check the data dependence via the heap access, i.e., the invocation I_2 is data dependent on the invocation I_1 if I_1 writes to the heap location read by I_2 and I_2 is reachable from I_1 in the inter-procedural control flow graph.

Atomicity. Atomicity commonly refers to the criterion of conflict serializability, which specifies that the atomic code region should not be interleaved by a conflicting access. According to the recent work [28], atomicity may also refer to the criterion of linearizability [28], which allows the atomic code region to be interleaved by a conflicting access but requires the atomic region to behave as if in serial settings in terms of input/response values. In either case, our inference captures the intended atomic composition correctly. However, different backend dynamic analyses are needed to check the correctness of the implementation.

For example, the class `AtomInteger` (Listing 3) from JDK 1.5+ composes the invocations of two atomic APIs `get` and `compareAndSet` to implement the new atomic functionality `getAndIncrement`. The atomicity of the composition refers to the linearizability criterion: the interleavings to the invocations are allowed, which lead to an extra iteration of the execution; the eventual effect of the code is to increase the `Integer` counter by one, the same as in the serial settings. Our inference identifies the intended atomic composition correctly, but the backend dynamic analysis would report the code as non-atomic as it violates the conflict serializability. A dynamic analysis designed to check the linearizability is needed to report properly for this example.

Listing 3. The `getAndIncrement` method

```
for(;;){
int current = get();
int next = current + 1;
if (compareAndSet(current, next))
 return current;
}
...
```

Limitation of Static Analysis. We rely on the static analyses to infer atomic sets (Section 3) and compute the side effects of the client-side invocations (Section 4). It is commonly known that static analyses are potentially unsound in the presence of dynamic loading mechanism, such as reflection, because the analyses may not observe code that will be loaded. Bodden et al. [2] mitigate the problem by monitoring the executions and recording the code loaded.

Furthermore, if an atomic set is only ever accessed by a single thread, the compositions of invocations that access it do not need to be atomic. However, static analyses cannot always determining whether the atomic set is accessed by multiple threads concurrently, so the analysis will be conservative and likely report such compositions as needing to be atomic. However, such compositions do not lead to bug reports by ICfinder: our dynamic analysis will not be able to exhibit non-atomic interleavings since only a single thread exists.

Limitation of Dynamic Analysis. We use the predictive analysis to search for non-atomic interleavings. No matter how effective the dynamic analysis is, it is not complete in general, i.e., it may not cover all the executions or all the buggy interleavings. We mitigate the problem by extensively applying the predictive analysis, aiming at finding as many non-atomic interleavings as possible.

8. EVALUATION

In this section, we aim at evaluating the effectiveness of the inference of atomic compositions, the efficiency of the dynamic checking, the quality of the final reports, and the validity of our high fidelity claim. The evaluation details are in Section 8.1, Section 8.2, Section 8.3 and Section 8.4 respectively.

We implement our approach as a tool ICfinder[2], and evaluate on large scale applications including Openjms, Lucene, Jigsaw, Derby, Tomcat. The benchmarks (shown in Table 1) that we use are from the Stamp [20] suite [3], the Dacapo [1] suite or the research work [5, 13] related to the atomicity. In the applications, the atomicity is required by the threads that access the shared resources, e.g., the shared physical connection in Openjms. 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.

8.1 Inference of Atomic Compositions

We apply the static analysis to infer the atomic compositions. The static analysis first discovers the library module, which provides the APIs for accessing the encapsulated atomic set and the client module, which invokes the library APIs to access the atomic set. It then finds the atomic compositions of the invocations at the client side with two key symptoms. The analysis is fully automated, without requiring extra manual specifications.

In the following, we present the details of the library/-client modules discovered by our analysis, then we study the effectiveness of our symptoms in identifying the atomic compositions.

Table 1 shows the total number of classes in the library modules and the client modules in Column LM and Column CM respectively, It also shows the number of atomic compositions (Column AC) identified by ICfinder and the number of the library APIs (Column API) involved in the atomic compositions.

According to the table, the atomic compositions, although occupying a small portion of the code base (around 1% for large applications), are too many for programmers to manage. For large applications, Lucene, Jigsaw, Derby and Tomcat, the number of atomic compositions ranges from 121 to 391. Given so many atomic compositions, manual reasoning of them is tedious and non-modular, easily leading to incorrect

[2]It stands for "the finder of incorrect compositions", which is publicly available: `http://www.cse.ust.hk/prism/AC`.

[3]As the original Stamp benchmarks are written in C language, we use the Java version provided by Demskey et al.`http://demsky.eecs.uci.edu/software.php`.

compositions. The large number of atomic compositions also necessitate the fully automated dynamic checking (Section 8.2).

Table 1. Metrics for static inference and dynamic checking

Benchmarks	LM	API	CM	AC	IC	$time$
Vacation	1	9	1	16	0	14
Labyrinth3D	2	4	1	5	0	8
Bayes	1	3	2	6	1	12
Cache4j	3	14	3	91	1	18
Tuplesoup	1	2	1	11	0	25
Specjbb	3	6	3	6	1	239
Jspider	2	4	3	7	0	43
Openjms	8	20	11	126	15	879
Lucene	15	91	35	391	2	142
Jigsaw	38	71	47	121	2	177
Derby	22	78	19	122	16	155
Tomcat	35	89	42	365	12	35
Avrora	2	2	2	7	0	43

Next, we study the effectiveness of our symptoms by comparing with the symptom used by the state of the art approach MUVI. We first briefly explain MUVI and our implementation of it. MUVI operates at the memory access level. It first learns what memory accesses appear together frequently and then treats the *frequent togetherness* as the symptom of atomicity. We extend it so that it operates at the invocation level. We first learn what atomic library APIs are frequently invoked together and then treat the frequent togetherness as the symptom of the atomic composition, i.e., the library APIs frequently invoked together are composed as atomic.

From the implementation view, MUVI applies a frequent itemset mining algorithm FPclose [9]: if two accesses acc_1 and acc_2 appear together for more than $minSupport$ times, and, each time acc_1 is present, acc_2 is present in higher than $minConfidence$ probability. We use the DCI_close algorithm [19], of which the Java implementation is publicly available [4]. DCI_close is similar to FPclose, except that it relies on one threshold, $minSupport$. In our experiment, we set the $minSupport$ threshold as 2, which means the APIs are frequently invoked together if they are invoked together in more than 2 methods. Higher threshold is possible, but "2" is sufficient for demonstrating the difference between our symptom and MUVI's symptom, as explained soon. Besides, MUVI requires the code distance threshold for determining the togetherness. As the threshold differs from application to application, it requires great efforts in the fine tuning, which limits the practical utility. We simply determine two invocations are together if they are in the same method.

In Table 2, we show the atomic compositions detected by MUVI and ICfinder in Column AC_{MUVI} and Column AC respectively. For comparison, we also show the atomic compositions found by ICfinder but missed by MUVI in Column Δ, and the atomic compositions found by MUVI but missed by ICfinder in Column Δ'.

According to Column Δ' of Table 2, ICfinder misses the minority (often less than 33%) of the atomic compositions found by MUVI. For four applications such as Specjbb and Avrora, ICfinder misses none of the atomic compositions found by MUVI. On the other hand, according to Column Δ, MUVI misses the vast majority of atomic compositions found by ICfinder, e.g., around 80% of the atomic compositions are missed for each large application.

We investigate these observations by comparing our two symptoms separately with the MUVI symptom. Besides, we investigate the atomic compositions found by MUVI but missed by ICfinder, as well as the false positives of ICfinder. In the following, we use ICfinder-USE or ICfinder-COMP to refer to the ICfinder which functions with only the USE or the complementation symptom adopted.

Effectiveness of the USE Symptom. The atomic compositions reported by ICfinder-USE are shown in Column AC_{USE}. For comparison, we show the atomic compositions found by ICfinder-USE but missed by MUVI in Column Δ_{USE}, and the atomic compositions found by MUVI but missed by ICfinder-USE in Column Δ'_{USE}.

According to Column Δ_{USE}, MUVI misses a lot of atomic compositions found by ICfinder-USE, i.e., 50%-100% atomic compositions are missed. This is due to the natural limit of the statistic-based approach: MUVI depends greatly on the frequency of the composition, however, many compositions appear only in one method, which gives insufficient support to MUVI. For example, the bug in Listing 7 is missed by MUVI. Note, higher $minSupport$ threshold makes MUVI miss more atomic compositions.

One interesting observation is that ICfinder-USE can also identify the atomic compositions involving the exception handling. Listing 4 shows the `multiplex` method from Openjms, which adopts the multiplex of the channels to speed up the message passing. The former invocation `addChannel()` registers the channels in the shared channel pool, an atomic set encapsulated inside the instance `_channels`. The latter invocation `disconnect()` is invoked in the presence of the exception to disconnect the subset of channels registered and connected. The two operations are expected to run atomically to preserve the consistency of status among the channels. ICfinder-USE identifies such atomic composition by identifying the control dependence between the two operations upon the exceptional control flow graph (the control flow graph with the exceptional edges modeled).

Listing 4. Code snippet of Openjms

```
multiplex()
{
  try{
      ...
      _channels.addChannel(localChannel);
      // register
      ...
  }catch(Exception e)
  { _channels.disconnect(); // shutdown
  }
}
```

Conversely, as illustrated by Column Δ'_{USE}, ICfinder-USE misses atomic compositions found by MUVI. One such atomic composition is already shown in Listing 2. ICfinder-USE misses the atomic compositions mainly because they do not exhibit the program dependence that ICfinder-USE relies on. ICfinder-COMP complements ICfinder-USE as ICfinder-COMP does not require the invocations in the atomic composition to be linked with the program dependence.

[4] `http://www.philippe-fournier-viger.com/spmf`

Table 2. Atomic compositions

Benchmarks	AC_{MUVI}	AC	Δ	Δ'	AC_{USE}	Δ_{USE}	Δ'_{USE}	AC_{COMP}	Δ_{COMP}	Δ'_{COMP}
Vacation	10	16	6	0	5	5	10	11	1	0
Labyrinth3D	3	5	3	1	1	1	3	4	2	1
Bayes	3	6	3	0	3	3	3	3	0	0
Cache4j	63	91	42	14	75	36	24	43	8	28
Tuplesoup	6	11	6	1	2	0	4	9	6	2
Specjbb	3	6	3	0	3	3	3	4	1	0
Jspider	9	7	0	2	5	5	2	7	5	0
Openjms	13	126	118	5	19	16	10	110	102	5
Lucene	245	391	269	123	251	228	222	156	47	136
Jigsaw	31	121	101	11	82	76	25	43	28	16
Derby	32	122	110	20	71	61	22	53	51	30
Tomcat	217	365	204	56	233	145	129	132	59	144
Avrora	0	7	7	0	2	2	0	7	7	0

ICfinder-USE may lead to false positives. Due to the conservativeness of static analysis, the invocations which access different runtime atomic sets may be judged as accessing the same atomic set. our backend dynamic checking filters out those false positives.

Effectiveness of the Complementation Symptom. ICfinder-COMP finds the atomic compositions without requiring the program dependence between the invocations. ICfinder-COMP adopts the complementation symptom, i.e., the invocations are inseparable in forming a combined atomic operation, i.e., they are executed together or skipped together in any execution.

We show the atomic compositions reported by ICfinder-COMP in Column AC_{COMP}, and the comparisons with MUVI in Column Δ_{COMP} and Column Δ'_{COMP}, in analogy to Column Δ_{USE} and Column Δ'_{USE}.

According to Column Δ_{COMP}, MUVI often misses more than half of the atomic compositions found by ICfinder-COMP, which is also due to the aforementioned limitation of the statistic-based approach. In Column Δ'_{COMP}, the five entries with "0" suggest that ICfinder-COMP can find all atomic compositions found by MUVI in five applications. We investigate the rest applications where ICfinder-COMP misses the atomic compositions found by MUVI.

On one hand, the vast majority (quantified in Section 8.4) of atomic compositions that ICfinder-COMP miss are false positives produced by MUVI. For the example in Listing 5, the invocation of library APIs `logTransactionState()` and `close()` do not form the atomic composition. However, MUVI, which uses the lexical adjacency as the symptom, reports it.

Listing 5. Code snippet of Openjms

```
1  logTransactionState()
   {
3    switch(_state.value)
      case OPENED:
5     {
        log.logTransactionState(_state);
7       break;
      }
9     ...
      case CLOSED:
11    {
        log.close();
13      break;
      }
15 }
```

On the other hand, some atomic compositions that ICfinder-COMP misses are true positives. Listing 6 shows one such atomic composition. The invocation of the library API `destroy()` is executed only if the application is configured to run in the `GC_SYNCHRONOUS` mode, while the invocation of the other library API `close()` is executed without the restriction. In the `GC_SYNCHRONOUS` mode, the two invocations may be expected to function atomically, while in other modes, only the former invocation is executed. As the invocations are not always executed together, ICfinder-COMP misses the atomic composition. Actually, to better express the atomicity intention, programmers could follow the coding paradigm in the comment, which treats the invocation of `close()` at line 9 and the invocation of `destroy()` at line 10 as a combined atomic operation. With such coding paradigm, ICfinder-COMP can successfully find the atomic composition.

Listing 6. Code snippet of Openjms

```
1  logTransactionState()
   {
3    log.close();
     if(_mode=GC_SYNCHRONOUS)
5    {
       log.destroy();
7    }
     // if(_mode=GC_SYNCHRONOUS)
9    // { log.close();
     //   log.destroy();}
11   // else
     //   log.close();
13 }
```

Finally, the atomic compositions reported by ICfinder-COMP may contain false positives. One common case is, the invocations of APIs `Log.open()` and `Log.close()` always match each other, and therefore are judged as the atomic composition. However, they are often executed non-atomically to allow the interleaving updates to the `Log` instance. Another common case is the invocations of APIs `Collection.get()` and `Collection.put()`. Programmers commonly use the former invocation to get an item and use the latter to put the item back after some local updates. If the invocations complement each other in every execution, ICfinder-COMP identifies the atomic composition of the invocations. However, the invocations may not form the atomic composition so

that the `Collection` instance allows the concurrent get/put operations for efficiency. Such false positives could be easily pruned based on pattern matching.

8.2 Dynamic Checking

Given the inferred atomic compositions, we check dynamically the incorrect compositions, i.e., the implementation that violates the atomicity of the composition. We run the dynamic analysis 20 times for each program and report the total number of incorrect compositions in Column *IC* (Table 1). Compared to the atomic compositions (Column *AC* in Table 2), the incorrect compositions are much fewer, i.e., less than 10% of atomic compositions are incorrectly implemented (except in the benchmarks, Bayes, Specjbb, Openjms). Two reasons account for it: (1) the programmers can well synchronize the majority of the compositions; (2) due to the nature limit of the dynamic analysis, ICfinder cannot cover all the paths and interleavings, therefore, misses some incorrect compositions. Our dynamic analysis allows programmers to focus on the small fraction of incorrect compositions instead of inspecting the large set of atomic compositions one by one.

We also evaluate the efficiency of ICfinder and show the analysis time (unit: msec) in Column *time* (Table 1). The analysis is very efficient, i.e., it usually finishes within 1 second, which owes to the optimization (Section 6) designed for checking atomic compositions.

8.3 Case Studies

In this section, we evaluate the quality of the final reports, i.e., whether they are helpful to programmers in exposing bugs. We focus on the large application Tomcat, which has an actively maintained mailing list. We confirm the bugs by either sending the posts or examining existing posts. Five out of twelve incorrect compositions are confirmed by developers as bugs. Considering that Tomcat is heavily used in around 250,000 sites [3] including the famous ones [37] such as Linkedin.com and Ebay.com, we believe finding the new bugs (or newly reported bugs) automatically is significant.

Tomcat is an open source software implementation of the Java Servlet technologies. The code in Listing 7 is from the class `JspServletWrapper`, which contains a field named `_theServlet`. The field together with the fields referenced by it form an atomic set. Accordingly, the library module includes the classes declaring these fields in the atomic set, e.g., the `JspServletWrapper` class and other classes.

Two atomic APIs provided by the library module are invoked: One API `getServlet` is invoked to update the fields in the atomic set to reflect the most recent change in the JSP file, the other API `service` is invoked to serve the incoming request from the JSP file. The latter invocation is data dependent on the former one via the heap access: the invocation of the method `service` uses the fields that are updated by the invocation of the method `getServlet`. Therefore, our static analysis identifies that the invocations form an atomic composition.

Listing 7. Code snippet of Tomcat

```
1 service(Request request, Response response
      ...)
  {
3 synchronized(this) { getServlet();}
  //_theServlet=...
5  if (mt_mode) {
       synchronized (this) {
7         _theServlet.service(request, response)
              ;
       }
9   }
  }
```

ICfinder then checks dynamically the implementation and finds it can be non-atomically interleaved, which leads to harmful behaviors. A remote invocation interleaves to destroy the instance referenced by `_theServlet`, making the state read at line 3 unavailable at line 7. As a consequence, line 7 may use a destroyed `_theServlet` instance to serve.

We report the bug to the developer. The developer confirms it as a real bug and fixes it in Tomcat 7.0.11 onwards (since Revision 1078409) [5].

Listing 8. Another code snippet of Tomcat

```
  removeAttribute(name)
2 {
    found = attributes.containsKey(name);
4   if (found) {
      ...
6     attributes.remove(name);
    }
8 }
```

Another confirmed Tomcat bug[6] is shown in Listing 8. The `ConcurrentHashMap` instance `attributes` and its referenced fields form an atomic set. Accordingly, the class `ConcurrentHashMap` and the referenced classes form the library module, which provides two atomic APIs, `containsKey` and `remove`.

The two APIs are invoked at the client method `removeAttribute` in the client module, i.e., the class `ApplicationContext`. As the latter invocation control depends on the former one, ICfinder identifies an atomic composition, which indicates the *invariant* that the remove operation is carried out only if the entry is present in the map.

By dynamically checking the implementation, ICfinder finds that the atomic compositions can be interleaved non-atomically by a remote invocation which removes the entry for `name`. The non-atomic interleaving violates the above invariant as the remove operation is carried out even if the entry for `name` is not present (it is removed by the remote invocation), which leads to the `NullPointerException`.

8.4 Assessment of High-fidelity Claim

We claim that our inference of atomic compositions achieves high fidelity, which means that it has few false negatives, i.e., it finds many of the places where atomic composition is required, while producing few false positives. While there is no direct way to compare our results with any notion of absolute truth, we argue below that we use reasonable approximations of truth. This claim has two aspects, which we address in turn.

Few False Negatives. The first interesting question is whether ICfinder misses many real atomic compositions. To answer this question, we approximate the truth by evaluating the compositions inferred by all techniques and checking if ICfinder misses real compositions found by other techniques.

[5] See the discussion in the mailing list: `http://mail-archives.apache.org/mod_mbox/tomcat-dev/201103.mbox/thread?2`

[6] Bug 53498: `https://issues.apache.org/bugzilla/show_bug.cgi?id=53498`

In order to get the maximal set of possible bugs, we here look at the compositions statically inferred rather than those confirmed by the dynamic analysis, in case the dynamic analysis fails to expose a real bug. While this does not represent an absolute truth, it is not clear how to find the missing bugs more comprehensively. Specifically, we investigate the application Lucene, where ICfinder misses the maximal number (123) of atomic compositions inferred by MUVI. By manually inspecting the atomic compositions, we find almost all atomic compositions that ICfinder misses are spurious. For example, the invocations of APIs `pauseAllThreads` and `resumeAllThreads` always pair each other and therefore are judged as an atomic composition by MUVI, but they do not actually require atomicity. The invocations of APIs `undeleteAll` and `deleteDocument` frequently pair each other in the same method and therefore are judged as an atomic composition, but they never run in the concurrent settings (i.e., only a single preparation thread executes them.) and therefore do not require atomicity. Overall, for Lucene, we did not find any genuine bugs that ICfinder missed.

Few False Positives. The second interesting question is whether the incorrect compositions found by ICfinder often represent real bugs. Note that our dynamic analysis reports only the compositions of which the atomicity violations are exhibited at runtime; the question now is whether the lack of atomicity leads to a genuine bug. We answer this question by evaluating which reports can be confirmed as bugs in Tomcat. We choose Tomcat because it is heavily used and so ought to have relatively hard-to-find bugs, and because the number of findings makes the manual validation practical. Recall from Table 1 that ICfinder finds 12 incorrect compositions. 5 of these 12 incorrect compositions uncover bugs, as confirmed by Tomcat developers[7].

In comparison, MUVI finds 53 incorrect compositions; however, with the same dynamic analysis as ICfinder, it uncovers only 1 confirmed bug. Thus, ICfinder has a much higher rate of actual bugs: 5/12=41.7% for ICfinder and 1/53=1.9% for MUVI. Bug reports accepted by the developers are the closest we can come to a gold standard of true positives, and, by this metric, a user of a heavily-used system like Tomcat would need to look at fewer than 3 reports to find a genuine bug with ICfinder.

9. RELATED WORK

Atomicity Intention Inference. SVD [40] is the first to infer the atomicity intention (or atomic regions). It infers the atomicity based on the single-variable serializability. Different from it, our work preserves the atomicity on multiple variables. AVIO [18] and AtomTracker [22] infer the atomic region as the longest unbreakable sequence in the correct runs. Kivati [4] applies the data flow analysis to identify consecutive accesses of a shared variable and treats them as in atomic regions. TransFinder [34] assumes conservatively all accesses of a shared variable form the same thread should be atomic. In addition, it applies the static analysis to remove the atomic regions which cannot be interleaved. The conservativeness of the static analysis degenerates its usefulness. MUVI [17] adopts the frequent togetherness as a symptom to infer the atomic regions. It also supports multi-variable atomicity.

[7] `http://www.cse.ust.hk/prism/AC`

Atomicity Violation Detection. Different from the above category of work, atomicity violation detection work focuses on checking whether the implementation allows non-atomic interleavings, with the atomicity intention correctly specified. As static analyses [6, 23] are incompetent in exploring the possible interleavings, very few static detection work of atomicity violations exists. A broad spectrum of dynamic analyses strive to explore the buggy interleavings both effectively and efficiently. Model checking [21, 29] systematically explores the interleavings. However, the exponentially large space makes it hard to scale to large applications. Active testing [26, 14] adopts the randomized scheduler to explore as many different interleavings as possible. It needs many runs to explore the interleaving space effectively, which may not be efficient. Predictive analyses [30, 13, 38] improve the efficiency by computing the possible buggy interleavings offline from a set of normal traces.

Atomicity on Multiple Variables. Besides Vaziri et al. [36], other researchers also observe and utilize the atomicity on multiple correlated variables. In the area of distributed system, consistency among multiple variables is a commonly-desired property. Weihl et al. [39] and Herlihy et al. [11] use the atomic objects to achieve the consistent execution and recovery. In the software transactional memory research, DSTM2 [10] and XSTM [25] support the usage of atomic objects.

10. CONCLUSION

Programmers often need to compose the atomic APIs to synthesize new atomic functionality. The compositions, expected to be atomic, may be implemented incorrectly as non-atomic. We design the static analysis which recognizes the atomic compositions with the symptoms capturing the key program logic for them, and customize the predictive analysis to find the incorrect compositions. Our evaluation on a set of large scale applications shows, the static analysis finds up to 391 atomic compositions for an application, while half would be missed by the previous statistic-based approach. The dynamic analysis runs efficiently for up to 1 second. Overall, our approach finds around 50 incorrect compositions, which are previously unknown. Our study on Tomcat shows that five out of twelve incorrect compositions are confirmed as bugs by the developers.

11. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and suggestions. We thank the Artifact Evaluation Committee (AEC) for validating our artifact. This research is supported by RGC GRF grant RGC622909 and RGC621912.

12. REFERENCES

[1] S. M. Blackburn and et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*.

[2] E. Bodden, A. Sewe, J. Sinschek, M. Mezini, and H. Oueslati. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE '11*.

[3] BuiltWith. Apache Tomcat Coyote Usage Statistics. `http://trends.builtwith.com/Web-Server/Apache-Tomcat-Coyote`, Feb. 2013.

[4] L. Chew and D. Lie. Kivati: fast detection and prevention of atomicity violations. In *EuroSys '10*.

[5] J. Dolby, C. Hammer, D. Marino, F. Tip, M. Vaziri, and J. Vitek. A data-centric approach to synchronization. *TOPLAS '12*.

[6] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03*.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *TOPLAS '1987*.

[8] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04*.

[9] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI '03*.

[10] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06*.

[11] M. Herlihy and J. Wing. Avalon: language support for reliable distributed systems. In *Symposium on Fault-Tolerant Computer Systems, 1987*.

[12] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *TOPLAS '1990*.

[13] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *ISSTA '11*.

[14] Z. Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE '10*.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM '1987*.

[16] P. Liu and C. Zhang. Axis: automatically fixing atomicity violations through solving control constraints. In *ICSE '12*.

[17] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP '07*.

[18] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII '06*.

[19] C. Lucchese. Dci closed: A fast and memory efficient algorithm to mine frequent closed itemsets. In *FIMI '04*.

[20] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for MultiProcessing. In *IEEE International Symposium on Workload Characterization '08*.

[21] M. Musuvathi, S. Qadeer, and T. Ball. Chess: A systematic testing tool for concurrent software.

[22] A. Muzahid, N. Otsuki, and J. Torrellas. Atomtracker: A comprehensive approach to atomic region inference and violation detection. In *MICRO '10*.

[23] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI '06*.

[24] R. H. Netzer and B. P. Miller. What are race conditions? - some issues and formalizations. *ACM Letters on Programming Languages and Systems '1992*.

[25] C. Noël. Extensible software transactional memory. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering '10*.

[26] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE '08*.

[27] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *PLDI '12*.

[28] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA '11*.

[29] O. Shacham, M. Sagiv, and A. Schuster. Scaling model checking of dataraces using dynamic information. In *PPoPP '05*.

[30] F. Sorrentino, A. Farzan, and P. Madhusudan. Penelope: weaving threads to expose atomicity violations. In *FSE '10*.

[31] W. N. Sumner, C. Hammer, and J. Dolby. Marathon: Detecting atomic-set serializability violations with conflict graphs. In *RV '11*.

[32] F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES, 1995*.

[33] T.J. Watson Libraries for Analysis. `http://wala.sourceforge.net`.

[34] G. Upadhyaya, S. P. Midkiff, and V. S. Pai. Automatic atomic region identification in shared memory spmd programs. In *OOPSLA '10*.

[35] R. Vallée-Rai and et al. Optimizing Java bytecode using the Soot framework: Is it feasible? In *CC '00*.

[36] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06*.

[37] W3Techs. Usage Statistics and Market Share of Tomcat for Websites. `http://w3techs.com/technologies/details/ws-tomcat/all/all`, Feb. 2013.

[38] C. Wang and M. Ganai. Predicting concurrency failures in the generalized execution traces of x86 executables. In *RV '11*.

[39] W. Weihl and B. Liskov. Implementation of resilient, atomic data types. *TOPLAS '1985*.

[40] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI '05*.