Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Finding Concurrency-Related Bugs using Random Isolation*

Nicholas Kidd[1], Thomas Reps[1,2], Julian Dolby[3], Mandana Vaziri[3]**

[1] University of Wisconsin; Madison, WI; USA. e-mail: {kidd,reps}@cs.wisc.edu
[2] GrammaTech, Inc.; Ithaca, NY; USA.
[3] IBM T.J. Watson Research Center; Hawthorne, NY; USA. e-mail: {dolby,mvaziri}@us.ibm.com

Received: date / Revised version: date

Abstract. This paper concerns automatically verifying safety properties of concurrent programs. In our work, the safety property of interest is to check for *multi-location data races* in concurrent Java programs, where a multi-location data race arises when a program is supposed to maintain an invariant over multiple data locations, but accesses/updates are not protected correctly by locks.

The main technical challenge that we address is how to generate a program model that retains (at least some of) the synchronization operations of the concrete program, when the concrete program uses dynamic memory allocation. Static analysis of programs typically begins with an abstraction step that generates an abstract program that operates on a finite set of abstract objects. In the presence of dynamic memory allocation, the finite number of abstract objects of the abstract program must represent the unbounded number of concrete objects that the concrete program may allocate, and thus by the pigeon-hole principle some of the abstract objects must be *summary objects*—they represent more than one concrete object. Because abstract summary objects represent multiple concrete objects, the program analyzer typically must perform *weak updates* on the abstract state of a summary object, where a weak update accumulates information. Because weak updates accumulate rather than overwrite, the analyzer is only able to determine *weak judgements* on the abstract state, i.e., that some property *possibly* holds, and not that it *definitely* holds. The problem with weak judgements is that determining whether an interleaved execution respects program synchronization requires the ability to determine *strong judgements*, i.e., that some lock is *definitely* held, and thus the analyzer needs to be able to perform *strong updates*—an overwrite of the abstract state—to enable strong judgements.

We present the *random-isolation abstraction* as a new principle for enabling strong updates of special abstract objects. The idea is to associate with a program allocation site *two* abstract objects, $r^\sharp$ and $o^\sharp$, where $r^\sharp$ is a non-summary object and $o^\sharp$ is a summary object. Abstract object $r^\sharp$ models a distinguished concrete object that is chosen at random in each program execution. Because $r^\sharp$ is a non-summary object—i.e, it models only one concrete object—strong updates are able to be performed on its abstract state. Because which concrete object $r^\sharp$ models is chosen randomly, a proof that a safety property holds for $r^\sharp$ generalizes to all objects modeled by $o^\sharp$.

We implemented the random-isolation abstraction in a tool called EMPIRE, which verifies *atomic-set serializability* of concurrent Java programs. (Atomic-set serializability is one notion of multi-location data-race freedom.) Random isolation allows EMPIRE to track lock states in ways that would not otherwise have been possible with conventional approaches.

1 Introduction

This paper concerns automatically verifying safety properties of concurrent programs. Although the problem is undecidable in general, in some cases undecidability can be side-stepped by using the following scheme for obtaining approximate answers:

- *Step 1:* Construct an abstraction of the program of interest using a modeling language for which safety checking is decidable.
- *Step 2:* Invoke the decision procedure to determine whether safety holds.
- *Step 3:* If the answer obtained in Step 2 is "yes" (safety holds), report "safe"; otherwise, report "unknown".

A safety-checking problem may be theoretically intractable from the standpoint of worst-case running time; however, in many cases answers can be obtained in a reasonable amount

---

Send offprint requests to: Nicholas Kidd

* Supported by NSF under grants CCF-0540955, CCF-0810053, and CCF-0904371, by ONR under grants N00014-09-1-0510 and N00014-09-1-0776, by ARL under grant W911NF-09-1-0413, and by AFRL under grants FA8750-06-C-0249 and FA9550-09-1-0279.

** current affliation: Google Inc

of time by incorporating fast algorithms and data structures for key steps of the decision procedure, or for key heuristics that it uses. Other work that adopts this approach includes symbolic model checking (Burch et al., 1990), SLAM (Ball and Rajamani, 2001), Moped (Schwoon, 2002), Lammich's work on the analysis of multi-threaded programs (Lammich and Müller-Olm, 2008), and the recent work on analyzing linked lists by Lahiri and Qadeer (2006), among others.

In our work, the safety property of interest is to check for *multi-location data races* in concurrent Java programs, where a multi-location data race arises when a program is supposed to maintain an invariant over multiple data locations, but accesses/updates are not protected correctly by locks. (The precise definition of multi-location data races used by our checking tool EMPIRE is presented in §2.) Multi-location data races generalize the traditional notion of data races (i.e., a *single-location data race* results from inconsistent coordination of accesses on a *single* memory location). The importance of checking for multi-location data races is underscored by a study conducted by Lu et al. (2008), which showed that roughly 30% of the concurrency bugs found in a set of concurrent open-source applications were due to multiple-memory-location problems. (Our approach can find classical single-location data races as well as multiple-location races.)

An example of a multi-location data race is found in one of the `Vector` constructors in Sun's JDK 1.4.2, which is illustrated by the code fragment show in Listing 1.[1] The class contains two fields `elementCount` and `elementData`, where `elementData` is an array of objects and `elementCount` is an integer field whose value specifies the number of valid components of a `Vector` object (i.e., the logical contents of a `Vector` object are `elementData[0]`–`elementData[elementCount − 1]`).

When a shared `Vector` object is accessed by concurrently executing threads T_1 and T_2, it is clearly desirable for T_1 to not be able to observe intermittent writes of T_2 to the two fields. Without proper synchronization, the invariant that `elementCount` specifies the logical boundaries of `elementData` could be violated. With an unfortunate scheduling of thread executions, such a violation could occur in the `Vector` constructor shown in Listing 1. In particular, the following interleaved execution would result in the invariant being violated:

1. T_1 begins executing the `Vector` constructor shown in Listing 1;
2. T_1 initializes `elementCount` to be `c.size()`;
3. T_2 removes all elements from the shared `Vector` referenced by the parameter `Collection c`;
4. T_1 initializes `elementData` with `Collection c` as modified by T_2.

At this point, if `c` was initially a non-empty collection, `elementCount` will now be non-zero, but the array `elementData` will have at each index position from `0` to `elementCount − 1` the `null` reference.[2]

[1] The snippet is a simplified version of (Vaziri et al., 2006, Fig. 2).

[2] The bug was first reported by Wang and Stoller (2006b)

Listing 1. java.util.Vector snippet.

```
public class Vector {
   Object[] elementData;
   int elementCount;

   Vector(Collection c) {
      elementCount = c.size();
      elementData = new
         Object[elementCount * 110 / 100];
      c.toArray(elementData);
   }
}
```

Our goal of checking for multi-location data races (such as the one just described) directly affects the choice of a modeling language. Specifically, the modeling language should support multiple threads of execution that communicate via shared memory, and that protect shared-memory accesses with locks. However, if we are not careful, the combination of the first two features alone could result in a modeling language for which safety checking is undecidable (Ramalingam, 2000).

In our work, decidability is maintained by carefully choosing which aspects of a concurrent Java program will be modeled precisely and which are abstracted away. The problems that were necessary for us to address can be summarized as follows: (1) How are data values modeled? (2) How are program locks modeled? (3) How are Java's reentrant locks modeled? (4) What is the algorithm for analyzing the resulting program model? Below we discuss each issue in turn.

Abstracting Values. The first insight is that multi-location data races are concerned with reads from and writes to fields of shared objects, and *not* with the values of those fields. Thus, a safe approximation abstracts away the values of all variables. Once the actual data values have been abstracted away, the reads and writes of an individual thread can no longer be affected by the reads and writes of other threads. Thus, the tight coupling between threads that typically results in analysis of concurrent programs (or program models) being undecidable has been removed. As in many program analyses, the price one pays is that the set of behaviors of the program model is necessarily an over-approximation of the set of behaviors of the (concrete) program, and hence one-sided answers are obtained: safe/possibly-unsafe rather than safe/definitely-unsafe.

Abstracting Locks. Once data values have been abstracted away, the resulting program contains no synchronization, which is obviously not adequate for checking properties that deal with program synchronization. In general, one cannot simply decide to model the "lock state" of every Java object o—i.e., whether the lock associated with o is in the *locked* or *unlocked* state—because Java programs contain dynamic memory allocation, which results in an *a priori* unbounded number of locks. To maintain decidability, we require a mechanism that is able to model the unbounded number of *concrete* locks—i.e., the set of all locks that are allocated by all dy-

namic concrete runs of the program—with a finite number of *abstract* locks. Unfortunately, the typical approach of using *summary objects* does not provide the precision necessary to reason about lock-based synchronization.

For *summary*-based analyses, an unbounded set of concrete objects are abstracted into a finite set of abstract objects. One common approach is to use the *allocation-site abstraction* of Jones and Muchnick (1982), where an abstract object $o^\sharp$ is defined for each static allocation site of the program (i.e., a `new` statement in Java), and the abstract state of $o^\sharp$ is a summary of all possible concrete objects that could be allocated from that site. It follows from the pigeon-hole principle that there must be at least one abstract object that represents more than one concrete object.

The difficulty is that when abstraction techniques summarize multiple objects into a single summary object, the analyzer is unable to perform strong updates (overwrites) when interpreting a lock or unlock operation. A strong update of an abstract object corresponds to a “group kill”: it represents a definite change in value to *all* concrete objects that the abstract object represents. Strong updates cannot generally be performed on summary objects because a (concrete) update only affects *one* of the summarized concrete objects. To be sound, an analyzer has to use weak updates (i.e., it has to accumulate lock states). In particular, it has to use an abstract lock state that represents $\{locked, unlocked\}$. Such an approach generally leads to significant loss of precision: an analysis will typically end up showing that a given summary lock is in the state $\{locked, unlocked\}$, which provides no information whatsoever.

Our second insight, which is the main technical contribution of the paper, is a novel abstraction technique called *random isolation*. The idea that we introduce is to analyze a *transformed* version of the program in which a set of user-specified allocation sites are transformed so that, during each concrete run of the program, at most one object r will be randomly tagged as being distinguished for that allocation site. By this means, an analyzer can keep the abstract representative for r (say $r^\sharp$) separate from a summary object that represents the rest of the objects allocated at the specified allocation site.[3] The advantage of random isolation is that because r is selected *randomly*, properties proved about its abstract representative $r^\sharp$ can be generalized to *all* objects that are potentially allocated at the same allocation site. Moreover, because each concrete run can have only *one* instance of r, $r^\sharp$ is not a summary object, which means that strong updates can be performed on the abstract state of $r^\sharp$—and hence its lock state can be tracked precisely. In practice, the analyzer will typically require an additional mechanism to ensure that a reference-type variable (e.g., `this` in Java) refers solely to $r^\sharp$, which can be difficult due to aliasing. We address this problem via a source-to-source transformation that injects expressions that perform a case analysis on what the reference variable points to (§6.1).

[3] Our convention is to use r for a randomly tagged concrete object, o for an arbitrary concrete object, and $r^\sharp$ resp. $o^\sharp$ for their abstract counterparts. When an allocation-site SITE is specified, we use the subscripted variants r_{SITE}, o_{SITE}, $r^\sharp_{\text{SITE}}$, and $o^\sharp_{\text{SITE}}$.

Reentrant Locks. EMPIRE checks concurrent Java programs that use reentrant locks (obtained via `synchronized` methods and `synchronized` blocks). Our third insight is that when modeling the locking structure of a program, one can (in some sense) ignore nested manipulations of the same lock. That is, only outermost lock/unlock pairs are truly important.

The straightforward way to encode a reentrant lock is to model it with a pushdown automaton (context-free language) that tracks the number of successive lock acquisitions on the stack. Instead, we apply a transformation that replaces the *context-free language* that describes a *reentrant* lock by a *regular language* that describes a *non-reentrant lock*. We call this transformation *language-strength reduction*, and it is performed in a completely accurate way (i.e., the model's behavior is preserved). Language-strength reduction is further discussed in §8.3.

Analyzing Abstract Programs. After program transformation and abstraction, the resultant program model contains a finite number of threads, shared memory locations, and shared locks. However, one still is left with the task of analyzing such a model. The model-checking engine on which EMPIRE was originally based employed a semi-decision procedure (Kidd et al., 2009a). The reason was that with (i) reentrant locks, (ii) an unbounded number of context switches, and (iii) an unbounded number of lock acquisitions, the problem appeared to be undecidable. The actions of each process are modeled by a context-free language; any model checker for multiple processes is dangerously close to the undecidable problem of determining whether the intersection of two context-free languages is empty. Surprisingly, after the problem has been transformed by the random-isolation and language-strength-reduction transformations, a decision procedure exists for the resulting simplified problem.

In essence, the couplings between different processes turned out to be sufficiently weak that the undecidability of checking emptiness of the intersection of two context-free languages did not come into play; however, that fact was obfuscated in a non-trivial way in the conditions of the original problem. We return to this discussion in §8 where an overview of the decision procedure is presented. (A full description of the decision procedure is available elsewhere (Kidd et al., 2011).)

Summary. The above sequence of ideas allowed us to create a tool that (i) can model concurrent Java programs quite faithfully, and (ii) check them for important kinds of concurrency problems: both *multi-location* and *single-location data races*. When we compared the performance of the decision procedure against the earlier semi-decision procedure (which represented the current state-of-the-art), we found that the decision procedure was 34 times faster overall (i.e., comparing the sum of the running times on all queries, with a 300-second timeout). Moreover, with the 300-second timeout threshold, for each

query where the semi-decision procedure timed out (roughly 49% of the queries), the decision procedure succeeded within the allotted time.

Outline. The remainder of this paper is organized as follows: §2 provides background material on atomic sets and AS-serializability, the notion of multi-location data race used in the paper. §3 presents an overview of EMPIRE, our tool for checking for multi-location and single-location data races. §4 presents EML, the EMPIRE modeling language. §5 motivates the need for a new abstraction by discussing how the commonly used allocation-site abstraction (Jones and Muchnick, 1982) is insufficient when checking for multi-location data races. §6 presents the random-isolation abstraction and describes an implementation of it. §7 presents a sound translation from a concurrent Java program abstracted using random isolation to an EML program. §8 describes the model-checking technique used to verify AS-serializability for an EML program. §9 presents the experimental evaluation. §10 discusses related work.

2 Atomic-Set Serializability

This section presents *atomic sets*—the mechanism to specify the fields over which an invariant holds—and *atomic-set serializability*—a data-centric correctness criterion that is used to define both single-location and multi-location data races. We present both atomic sets and atomic-set serializability (AS-serializability) via the example program shown in Listing 2. (In Listing 2, assume that all methods are public. The Java keyword `public` has been omitted to save space. Moreover, the Java keyword `synchronized` has been abbreviated as `sync`.)

An *atomic set* is defined with respect to a Java class `C` and specifies that an invariant holds over a subset of the fields defined by `C`. `C` may define as many atomic sets as fields, however, each field is allowed to be a member of at most one atomic set (i.e., the atomic sets are disjoint).

Example (Atomic Sets). Listing 2 presents a simple Java program that defines a class `Stack` that implements a stack data structure using an array `Stack.data` for storage and an integer `Stack.count` whose current value is the top-of-stack index into the array `Stack.data`. (For now, ignore the class `SafeWrap`.) The annotations `@atomic(S)` on fields `Stack.data` and `Stack.count` indicate that those fields are members of the atomic set "`S`". The (unspecified) invariant over `data` and `count` is that the current value of `count` is the index of the position in `data` where the top-of-stack item is stored. The existence of this relationship is reflected by both fields being members of the atomic set "`S`".

Associated with atomic sets are *units of work*; a unit-of-work annotation is a programmer assertion that a method guarantees to maintain the invariant of an atomic set when executed serially (i.e., the unit-of-work is executed to completion and all other threads wait for it to complete). We call a unit of work that writes to all members of an atomic set a *write-complete* unit of work.

Listing 2. Concurrent Stack program.

```
class Stack {                                              1
 public static final int MAX=10;                           2
 @atomic(S) Object[] data = new Object[MAX];               3
 @atomic(S) int count = -1;                                4
                                                           5
 @atomic sync Object pop(){                                6
  Object res = data[count];                                7
   data[count--] = null;                                   8
  return res;                                              9
 }                                                         10
 @atomic sync void push(Object o) {                        11
  data[++count] = o;                                       12
 }                                                         13
 @atomic sync int size() {                                 14
  return count+1;                                          15
 }                                                         16
 @atomic sync replaceTop(Object o) {                       17
  pop();                                                   18
  push(o);                                                 19
 }                                                         20
 static Stack makeStack() {                                21
  return new_SITE Stack();                                 22
 }                                                         23
}                                                          24
class SafeWrap {                                           25
 @atomic sync Object popwrap(@atomic Stack s) {            26
  return (s.size() > 0) ? s.pop() : null;                  27
 }                                                         28
 static SafeWrap makeSafeWrap() {                          29
  return new SafeWrap();                                   30
 }                                                         31
                                                           32
 static void main(String[] args){                          33
  Stack stack = Stack.makeStack();                         34
  stack.push(new Integer(1));                              35
  new Thread("1") {                                        36
   makeSafeWrap().popwrap(stack);                          37
  }                                                        38
  new Thread("2") {                                        39
   makeSafeWrap().popwrap(stack);                          40
  }                                                        41
 }                                                         42
}                                                          43
```

Example (Units of Work). In Listing 2, the units of work are methods that have the `@atomic` annotation (e.g., the method `Stack.pop`). The write-complete units of work are the methods `Stack.pop`, `Stack.push`, and `Stack.replaceTop`.

The correctness criterion associated with atomic sets is *atomic-set serializability* (AS-serializability). AS-serializability is a generalization of *serializability*. In general, a code region that should appear to execute atomically forms a transaction (i.e., represents a logically-atomic operation). An interleaved program execution is then *serializable* if it is equiv-

alent to an execution in which the transactions are executed in some serial order. For AS-serializability, the atomic-code regions (i.e., transactions) are the unit-of-work methods. An execution is said to be AS-serializable if its projection on each atomic set is serializable, where the projection of an execution with respect to an atomic set S removes all reads and writes to memory locations that are not members of S. AS-serializability is a generalization of serializability because serializability can be obtained by defining all of memory to be in one atomic set.[4]

Example (AS-serializable). Consider the method `Stack.replaceTop`, whose body merely invokes the methods `Stack.pop` and `Stack.push`. Each of the three methods are `synchronized` methods, and the atomic set with respect to which they must execute atomically is "S". Because the methods are synchronized, no other thread can access the fields that are members of "S" until `Stack.replaceTop` completes execution. Thus, all executions of `Stack.replaceTop` must be AS-serializable.

Atomic sets can be dynamically extended to include the atomic sets of method parameters, which are referred to as *unitfor* parameters. This is specified by an `@atomic` annotation on a method parameter.[5] Dynamic extension of an atomic set captures the fact that if a unit of work is dependent on a method parameter, execution of that unit of work must (appear to) be atomic for the atomic sets of the parameter as well.

Example (Unitfor Parameters). Class `SafeWrap` does not define an atomic set; however, method `popwrap` is defined as a unit of work and parameter `Stack s` as a unitfor parameter. Thus, `popwrap` should execute atomically with respect to the units of work of class `Stack`, namely the atomic set S.

2.1 AS-serializability Violations

An execution that is not AS-serializable is said to have an AS-serializability violation. A result from Vaziri et al. (2006) is that if all units of work are write-complete, then AS-serializability violations can be completely characterized by a set of fourteen problematic access patterns. Tab. 1 presents the fourteen problematic access patterns.[6] Each pattern consists of a sequence of reads (R) and writes (W) to an atomic-set member l, or to two atomic-set members l_1 and l_2. Problematic access patterns 1 through 5 capture single-location data-consistency errors, while problematic access patterns 6 through 14 capture multi-location data-consistency errors. Each memory access occurs during a unit of work u; the subscript on the memory access denotes which unit of work it belongs to. For example, problematic access pattern 1 is defined as "$R_u(l)\ W_{u'}(l)\ W_u(l)$", which describes an AS-serializability violation that involves an atomic-set member l. The first read and last write belong to unit of work u. The intervening write belongs to a unit of work u' that is executed by another thread.

To give a concrete illustration of an AS-serializability violation, let us return to the program in Listing 2. The method `Stack.pop` performs no safety checking. That is, `Stack.pop` does not verify that the stack is non-empty before attempting to remove the top-of-stack item. If `Stack.pop` is invoked on an empty stack, then the field `Stack.count` will be -1. The array access to the field `Stack.data` will result in an `ArrayIndexOutOfBoundsException` being raised because `Stack.count` has the value -1.

The class `SafeWrap` defined below `Stack` in Listing 2 addresses this oversight by defining the method `SafeWrap.popwrap`, which first checks that the parameter `Stack s` is not empty before invoking the method `Stack.pop`. For the class `SafeWrap`, the method `SafeWrap.popwrap` is a unit of work, indicated by the `@atomic` annotation on the method, and the parameter "`Stack s`" is a unitfor parameter, indicated by the `@atomic` annotation on parameter `s`. (Note that the `@atomic` annotation provides a specification of the desired behavior, and not an implementation of it.) Recall that the atomic sets of a unitfor parameter are dynamically incorporated into the atomic sets of the invoking object, i.e., the `this` parameter, for the duration of a unit of work. Because class `SafeWrap` does not define any atomic sets, the atomic set with respect to which `SafeWrap.popwrap` must execute atomically is the atomic set S defined by class `Stack`.

Unfortunately, the attempt to "harden" the code by adding error checking has introduced an AS-serializability violation. The problem is that the program synchronizes on the wrong object. In this case, the method `SafeWrap.popwrap` has the `synchronized` annotation, whereas the method should have been written to synchronize on the parameter `Stack s`. Thus, the interleaved execution shown in Fig. 1 is valid.

The interleaved execution shown in Fig. 1 contains problematic access pattern 12. The data accesses that are involved in the pattern are underlined in Fig. 1. Because of the AS-serializability violation, the method `SafeWrap.popwrap` can raise an `ArrayIndexOutOfBoundsException` even when it is invoked on a non-empty stack. The interleaved execution shown in Fig. 1 illustrates how that can happen. Initially, the stack contains one item. Thread T_1 begins execution and checks that the stack is non-empty by invoking `Stack.size`. The check succeeds, and so T_1's next action is to invoke `Stack.pop`. Before doing so, thread T_2 successfully executes `SafeWrap.popwrap`, which removes the item from the stack, leaving it empty. When T_1 resumes execution, it invokes `Stack.pop` on an empty stack, which raises an exception. The point at which the exception is raised is denoted by the symbol ✠ at the end of thread T_1's execution sequence.

---

[4] Defining serializability in terms of AS-serializability requires the ability for atomic sets to span class definitions.

[5] Vaziri et al. (2006) use the annotation "unitfor" to mark method parameters that should be dynamically absorbed into an atomic set.

[6] The patterns in Tab. 1 appear in Vaziri et al. (2006) and Hammer et al. (2008).

Id	*Problematic Access Pattern*				*Description*
1.	$R_u(l)$	$W_{u'}(l)$	$W_u(l)$		Value read is stale by the time an update is made in u.
2.	$R_u(l)$	$W_{u'}(l)$	$R_u(l)$		Two reads of the same location can yield different values in u.
3.	$W_u(l)$	$R_{u'}(l)$	$W_u(l)$		An intermediate state is observed by u'.
4.	$W_u(l)$	$W_{u'}(l)$	$R_u(l)$		Value read may not be the same as the one written last in u.
5.	$W_u(l)$	$W_{u'}(l)$	$W_u(l)$		Value written by u' can be lost.
6.	$W_u(l_1)$	$W_{u'}(l_1)$	$W_{u'}(l_2)$	$W_u(l_2)$	Memory can be left in an inconsistent state.
7.	$W_u(l_1)$	$W_{u'}(l_2)$	$W_{u'}(l_1)$	$W_u(l_2)$	same as above
8.	$W_u(l_1)$	$W_{u'}(l_2)$	$W_u(l_2)$	$W_{u'}(l_1)$	same as above.
9.	$W_u(l_1)$	$R_{u'}(l_1)$	$R_{u'}(l_2)$	$W_u(l_2)$	State observed can be inconsistent.
10.	$W_u(l_1)$	$R_{u'}(l_2)$	$R_{u'}(l_1)$	$W_u(l_2)$	same as above
11.	$R_u(l_1)$	$W_{u'}(l_1)$	$W_{u'}(l_2)$	$R_u(l_2)$	same as above.
12.	$R_u(l_1)$	$W_{u'}(l_2)$	$W_{u'}(l_1)$	$R_u(l_2)$	same as above.
13.	$R_u(l_1)$	$W_{u'}(l_2)$	$R_u(l_2)$	$W_{u'}(l_1)$	same as above.
14.	$W_u(l_1)$	$R_{u'}(l_2)$	$W_u(l_2)$	$R_{u'}(l_1)$	same as above.

Table 1. The fourteen problematic access patterns. Patterns 1–5 involve a single memory location; patterns 6–14 involve a pair of memory locations.

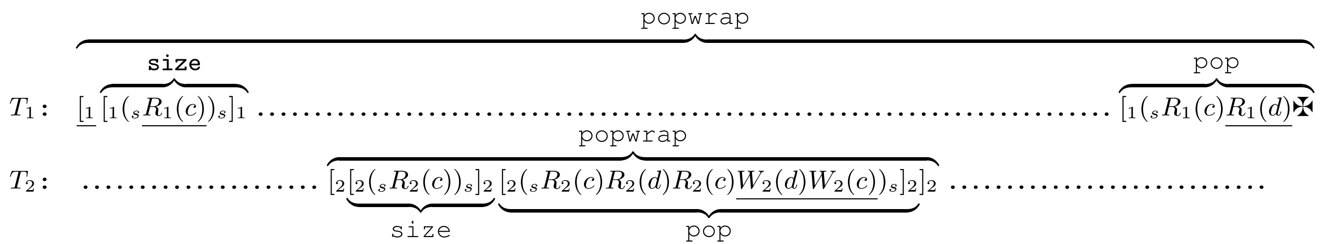

Fig. 1. An interleaved execution of thread T_1 and T_2 that contains an AS-serializability violation. R and W denote a read and write access, respectively. c and d denote fields `count` and `data`, respectively. "[" and "]" denote the beginning and end, respectively, of a unit of work. The subscripts "1" and "2" are thread ids. "$(_s$" and "$)_s$" denote the acquire and release operations, respectively, of the lock of `Stack s` that is the input parameter to `SafeWrap.popwrap`.

3 EMPIRE

EMPIRE is a tool to verify that errors such as the one described at the end of §2 do not occur in a concurrent Java program Prog. EMPIRE returns answers of the form "the problematic access pattern p is definitely not present" or "the problematic access pattern p may be present", where p is an integer in the range one through fourteen that specifies the particular problematic access pattern of interest (cf. Tab. 1). EMPIRE uses abstraction to generate an abstract program $\mathsf{Prog}^\sharp$ such that the set of behaviors of $\mathsf{Prog}^\sharp$ is a *sound over-approximation* of the set of behaviors of Prog. The challenge is to define a finite-data abstraction such that $\mathsf{Prog}^\sharp$ is able to disallow certain thread interleavings by modeling the synchronization of Prog's processes, and yet analysis of $\mathsf{Prog}^\sharp$ remains decidable. These two properties are closely intertwined: if the program model becomes more precise it likely implies that a more powerful decision procedure will be required.

The EMPIRE toolchain is shown in Fig. 2. The input to EMPIRE is a concurrent Java program Prog along with a set of user-specified allocation sites. The allocation sites direct EMPIRE to (attempt to) verify that all executions are AS-serializable with respect to the objects that can be allocated from those sites. We will generally refer to a specified allocation site by SITE.

Example (Specified Allocation Site). In Listing 2, the allocation site of interest is signified by the subscripted $\texttt{new}_{\text{SITE}}$ statement on line 22.

EMPIRE's abstraction process proceeds in two stages. First, a source-to-source program transformation (abbreviated as "Src-to-Src" in Fig. 2) bootstraps the implementation of the random-isolation abstraction. Second, the transformed program is abstracted via random-isolation abstraction ("RIA" in Fig. 2) to generate an EML program. From one EML program, multiple queries are generated for the IPAMC model checker that implements our decision procedure. The input to IPAMC is a *multi-pushdown system* and an *indexed phase automaton* that specifies the safety property. We defer the definitions of these last two entities until §8.

Before getting to random isolation, we begin by defining the EMPIRE Modeling Language (EML). Defining EML first serves to motivate the need for a new abstraction technique.

4 EML

An EML program EProg consists of (i) a finite set of shared-memory locations S_{Mem}; (ii) a finite set of reentrant locks S_{Locks}; and (iii) a finite number of concurrently executing processes S_{Procs}.

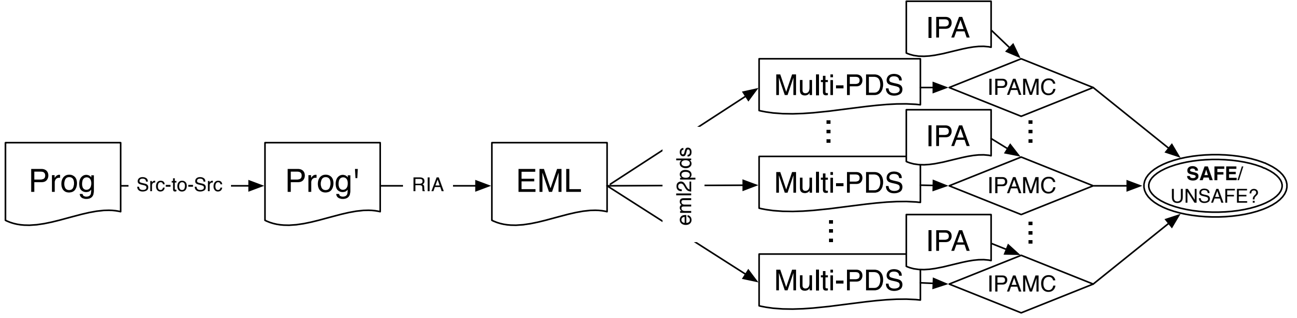

Fig. 2. The EMPIRE toolchain. "EML" stands for EMPIRE Modeling Language (§4); "RIA" stands for random-isolation abstraction (§6); "Multi-PDS" stands for multi-pushdown system (§8); "IPA" stands for indexed-phase automaton (§8); and "IPAMC" is our model checker for multi-PDSs (Kidd et al., 2011).

Memory. An EML shared-memory location m is an abstract memory location: abstract reads and writes can be made on m; however, EML does not have a notion of a value held by m. The lack of values stored at shared-memory locations is an artifact of EMPIRE's goal of verifying AS-serializability, which is a property of the order of interleaved reads and writes of an application, and not of the values read and written. Moreover, abstracting away values is required to ensure that the analysis of an EML program remains decidable.

Locks. An EML lock is reentrant, meaning that the lock can be reacquired by the EML process that currently owns the lock, and also that the lock must be released the same number of times as it was acquired to become free. EML restricts the acquisition and release of an EML lock to occur within the body of a function, i.e., an EML lock cannot be acquired in a function f and released in another function f'. In addition, the acquisition of multiple EML locks by an EML process must be properly nested: an EML process must release a set of held locks in the order opposite to their acquisition order. The two restrictions are naturally fulfilled when EML is used to model synchronized methods and blocks of a Java thread.[7]

An EML lock l can begin in an "unallocated" state and will remain unallocated until an EML process makes a transition on an edge alloc l (see Tab. 2). Lock allocation is used to signify whether or not the randomly-isolated object that is associated with l has been allocated. In §6.1, which discusses the implementation of random-isolation, alloc l_{SITERI} is used to implement (in EML) the setting of the global flag $\mathcal{F}_{\text{SITERI}}$. EML does not have "objects" and object allocation does not occur—there is a finite set of shared memory locations S_{Mem}. The choice of the word alloc reflects the fact that, in the concrete Java program, a randomly-isolated object needs to be allocated before its lock can be acquired. Moreover, it can be allocated at most one time. To summarize, alloc l is a semantic action in the EML program; it denotes that the EML execution models a path in the *concrete* Java program that allocates a randomly-isolated (concrete) object.

[7] Java programs that use the synchronization primitives provided by the `java.util.concurrent` library need not satisfy the restriction that lock usage is properly nested. To analyze such programs, one would need to bypass EML and generate an IPAMC query (§8) directly—the caveat being that any results obtained from analyzing an IPAMC query that does *not* use properly nested locks is meaningless because the input query lies outside the domain of allowable queries.

Labels	Semantics
call f	invoke function f
read m	read from memory location $m \in S_{\mathsf{Mem}}$
write m	write to memory location $m \in S_{\mathsf{Mem}}$
alloc l	allocate the EML lock $l \in S_{\mathsf{Locks}}$
lock l	acquire the EML lock $l \in S_{\mathsf{Locks}}$
unlock l	release the EML lock $l \in S_{\mathsf{Locks}}$
unitbegin	begin a unit of work
unitend	end a unit of work
start Proc	start EML process Proc
skip	a statement whose semantic action is not modeled

Table 2. The edge labels Labels of an EML flow graph that represents an EML function and their corresponding semantics.

Processes. An EML process Proc is defined by a set of (possibly) recursive functions, one of which is designated as the `main` function of the process. An EML function f is defined by a *labeled-flow graph* $\mathcal{G}_f$. A labeled-flow graph is a tuple $\mathcal{G}_f = (\mathsf{Nodes}_f, \mathsf{Labels}, \mathsf{Edges}_f, n_{\mathsf{entry}}, n_{\mathsf{exit}})$, where:

- Nodes_f is a set of nodes;
- Labels is defined with respect to the set of memory locations S_{Mem} and set of locks S_{Locks} as shown in Tab. 2;
- $\mathsf{Edges}_f \subseteq \mathsf{Nodes}_f \times \mathsf{Labels} \times \mathsf{Nodes}_f$ is a set of edges; $n_{\mathsf{entry}} \in \mathsf{Nodes}_f$ is the distinct entry node; and
- $n_{\mathsf{exit}} \in \mathsf{Nodes}_f$ is the distinct exit node.

An edge $e = (n, a, n') \in \mathsf{Edges}_f$ models the flow of control from node n to node n'. Non-determinism is introduced by having multiple outgoing edges from the same node. (EML programs have only non-deterministic branches.) The label $a \in \mathsf{Labels}$ represents a semantic action that the EML process performs when transferring control from node n to node n'. EML supports the labels listed in Tab. 2.

An edge labeled "start Proc" starts the EML process named Proc. This is used to model the fact that when a Java program begins, only one thread is executing its `main` method, and all other threads cannot begin execution until they have been started by an already executing thread.

The design of EML is strictly motivated by the desire to detect AS-serializability violations, and thus we do not give a formal specification of its semantics. Instead, the meaning of an EML program is defined by its translation into a set of IPAMC queries. In some sense, IPAMC is a virtual machine for running an EML program and monitoring its behavior with respect to indexed phase automata (IPAs, which will be defined in §8.1). The translation from EML to a multi-PDS (§8) is viewed as "compilation", and IPAs for the set of fourteen problematic access patterns define a language of disallowed program behaviors. An EML program is compiled into a multi-PDS, and, in effect, the decision procedure implemented by IPAMC exhaustively runs the EML program on all possible inputs and checks whether any of the runs is in the language of problematic access paths. We defer discussion of the details of this process until §8, and explain in §5–§7 how an EML program is generated. IPAMC itself is structured as a tool for checking an EML program with respect to an arbitrary IPA, and hence could be applied to problems other than detecting AS-serializability violations.

5 The Allocation-Site Abstraction

The definition of EML in §4 makes explicit the constraints that the result of program abstraction should satisfy: there must be a finite number of shared memory locations, a finite number of threads, and a finite number of reentrant locks that are acquired and released in a properly nested fashion. A concurrent Java program, however, will typically only satisfy the requirement that locks are acquired and released in a properly nested fashion (synchronized methods naturally satisfy this requirement).[8]

To address the incompatibilities between a concrete Java program and EML, we use program abstraction, and, in particular, the *random-isolation abstraction*. Before presenting the random-isolation abstraction, we motivate the need for a new abstraction by first discussing why the *allocation-site abstraction* (Jones and Muchnick, 1982)—a commonly used technique for modeling the unbounded set of dynamically allocated concrete objects with a finite set of abstract objects—is unsatisfactory. Specifically, we highlight the allocation-site abstraction's limitations when analyzing a concurrent program that uses an unbounded number of dynamically-allocated locks. Intuitively, the reason why locks pose such a problem is that modeling synchronization in the abstract program requires definite (must) information—i.e., information that an abstract lock, and hence all concrete locks represented by the abstract lock, is definitely in the *locked* state. When using the allocation-site abstraction, an analysis typically only has indefinite (may) information available. We make these concepts more concrete in the following discussion.

[8] We assume that an analyzed Java program is restricted to use `synchronized` methods and blocks and not other synchronization mechanisms. In particular, the non-syntactically restricted locks provided by the `java.util.concurrent` package are not used.

Given an allocation site $\textsc{Site}$ for class T, let $\mathit{Conc}(\textsc{Site})$ denote the set of all concrete objects of class T that can be allocated at $\textsc{Site}$. The allocation-site abstraction uses a single abstract object $o^\sharp_{\textsc{Site}}$ to summarize all of the concrete objects in $\mathit{Conc}(\textsc{Site})$. When the size of $\mathit{Conc}(\textsc{Site})$, denoted by $|\mathit{Conc}(\textsc{Site})|$, is greater than 1, the abstract object $o^\sharp_{\textsc{Site}}$ is referred to as a *summary object*. Thus, for each field f defined by T, field $o^\sharp_{\textsc{Site}}.f$ is a summary field for the set of fields $\{o.f \mid o \in \mathit{Conc}(\textsc{Site})\}$. Because the program has a finite number of program points, and each class defines a finite number of fields, this results in a finite-data abstraction.

Example (Allocation-Site Abstraction). There are five allocation sites in Listing 2: line 22 allocates a `Stack` object; line 30 allocates a `SafeWrap` object; line 35 allocates an `Integer` object; and lines 37 and 40 allocate `Thread` objects T_1 and T_2, respectively. All allocation sites except the one on line 30 are executed exactly one time for all executions of the program shown in Listing 2. The allocation site on line 30 allocates two concrete objects because the method `SafeWrap.makeSafeWrap` is invoked once by thread T_1 and once by thread T_2.

The allocation-site abstraction would define the five abstract objects $o^\sharp_{22}$,$o^\sharp_{30}$, $o^\sharp_{35}$,$o^\sharp_{37}$, and $o^\sharp_{40}$, where abstract object $o^\sharp_i$ represents all concrete objects that could be allocated at the allocation site on line i. For the program in Listing 2, the abstract objects $o^\sharp_{22}$,$o^\sharp_{35}$, $o^\sharp_{37}$, and $o^\sharp_{40}$ each represent a singleton set because the program only executes the associated allocation statement once (as discussed above). However, the abstract object $o^\sharp_{30}$ represents two concrete objects because the method `SafeWrap.makeSafeWrap` is invoked twice, once by threads T_1 and T_2, respectively.

For the allocation-site abstraction to be sound, an analysis generally has to perform *weak updates* on each summary object. That is, information for the summary object must be *accumulated* rather than overwritten. In contrast, a *strong update*—the alternative to a weak update—*overwrites* an abstract object's abstract state. Such an overwrite corresponds to a "group kill" when the abstract object is a summary object. Thus, a strong update of the abstract state generally can only be performed when the analysis can prove that there is at most one object allocated at $\textsc{Site}$, i.e., $|\mathit{Conc}(\textsc{Site})| = 1$. For the program in Listing 2, strong updates could be performed on all abstract objects except $o^\sharp_{30}$ because abstract object $o^\sharp_{30}$ represents two concrete `SafeWrap` objects. Note that an interprocedural analysis would be required to determine that the abstract object $o^\sharp_{22}$—the abstract object representing objects allocated by the `Stack.makeStack` method—represents a singleton set. The problem is that $o^\sharp_{22}$ is used to allocate *all* `Stack` objects, and thus an interprocedural analysis would be required to prove that the method `Stack.makeStack` is invoked exactly one time for *all* executions of the program. Because of this difficulty, analyses that make use of the allocation-site abstraction typically resort to the assumption that every abstract object is a summary object.

An AS-serializability violation is defined in terms of reads and writes to the fields of the T object allocated at a specified

allocation site SITE. Thus, to detect an AS-serializability violation, EMPIRE must track reads and writes to these fields. In Listing 2, the allocation site of interest is on line 22. This is denoted by the subscripted $\texttt{new}_{\text{SITE}}$ statement. The allocation-site abstraction is a sound over-approximation for modeling reads and writes because a read from (write to) the abstract field $o^\sharp_{\text{SITE}}.f$ corresponds to a possible read from (write to) $o.f$, for one concrete object o in *Conc*(SITE). For the progam shown in Listing 2, the reads and writes of interest are to the fields `Stack.data` and `Stack.count`. The @`atomic(S)` annotation on the two fields specifies that each field is a member of the atomic set S.

Checking for AS-serializability violations also requires EMPIRE to model program synchronization. If EMPIRE did not model synchronization operations, it would likely be useless because the number of abstract executions would grossly over-approximate the number of valid concrete executions, which would typically result in a large number of false positives. Modeling Prog's synchronization is accomplished by defining EML locks. There are two possibilities for defining the semantics of an EML lock:

1. The first possibility is to interpret a lock acquire as a *strong update*, i.e., the program has definitely acquired a particular lock. This would correspond to acquiring the locks of *all possible* instances in *Conc*(SITE), which in most circumstances—including the one here—would be unsound. In the example of Listing 2, this interpretation of locking combined with the allocation-site abstraction would preclude the interleaved program execution shown in Fig. 1 that contains the bug, because the two `SafeWrap` objects would effectively get the same lock, and the two `SafeWrap.popwrap` methods would execute without interleaving.
2. The second possibility for defining the semantics of EML locks is to interpret a lock acquire as a *weak update*, i.e., the program *may* have acquired a particular lock. A weak update would leave the lock in a "possibly-held" state. When an EML process Proc attempts to acquire a lock that is in the "possibly-held" state, two cases must be considered.
   (a) The lock is actually held by another EML process Proc′ and thus Proc must block until Proc′ releases the lock.
   (b) The lock is not held by another EML process Proc′ and thus Proc may acquire the lock.

   This semantics is sound because all possible cases are considered. However, the overall effect of this semantics is, in essence, equivalent to an EML program without locks because the program can never reason definitively whether or not a lock is actually held. That is, in the abstract program a thread is always able to acquire a lock, which in essence means that the abstract program operates as if there are no synchronization constraints. In general, this possibility would greatly increase the number of false positives. For instance, in the example of Listing 2, if we were to fix the code by adding an additional synchronization block on the parameter `Stack s` inside the body of `SafeWrap.popwrap`, the analysis would still report a bug because locking behavior was modeled imprecisely.

In summary, neither of the two possible semantics for locks with the allocation-site abstraction is satisfactory: the first leads to an unsound analysis and the second leads to an analysis that would be useless in practice. Thus, a new abstraction mechanism is required, one that is sound, but allows strong updates to be performed on at least some of the abstract locks (so that the analysis can (partially) reason about program synchronization).

6 Random-Isolation Abstraction

To address the deficiencies of using the allocation-site abstraction, we developed a new abstraction technique, *random-isolation*, which is a novel extension of allocation-site abstraction. The random-isolation abstraction is motivated by the following observation:

Observation 1. *The concrete objects that can be allocated at a given allocation site* SITE*, Conc*(SITE)*, cannot be distinguished by the allocation-site abstraction.*

Obs. 1 states that if one chooses to isolate a *random concrete* object r_{SITE} from *Conc*(SITE), the allocation-site abstraction would not be able to distinguish the randomly-chosen concrete object from any of the other concrete objects that are represented by $o^\sharp_{\text{SITE}}$.

The random-isolation abstraction leverages Obs. 1 by randomly isolating one of the concrete objects allocated at allocation site SITE and tracking it specially in the abstraction. Whereas allocation-site abstraction would use one summary object $o^\sharp_{\text{SITE}}$ to represent all concrete objects *Conc*(SITE) that can be allocated at SITE, random isolation uses two objects: one summary, $o^\sharp_{\text{SITE}}$, and one non-summary, $r^\sharp_{\text{SITE}}$. The object $r^\sharp_{\text{SITE}}$ is a non-summary object because it alone represents the randomly-isolated concrete object r_{SITE}. Because $r^\sharp_{\text{SITE}}$ is a non-summary object, it is safe to perform strong updates to its (abstract) state, which gives us *Random-Isolation Principle 1*.

Random-Isolation Principle 1 (Updates) *Let* $r_{\text{SITE}} \in$ *Conc*(SITE) *be a randomly-isolated concrete object. Because* r_{SITE} *is modeled by a special abstract object* $r^\sharp_{\text{SITE}}$*, the random-isolation abstraction enables an analysis to perform strong updates on the abstract state of* $r^\sharp_{\text{SITE}}$.

Random isolation also provides a powerful methodology for proving properties of a program, which is captured by following *Proofs Principle*. In plain English, to establish that a universally quantified safety property ϕ holds for all objects $o_{\text{SITE}} \in$ *Conc*(SITE), it suffices to establish the property for only the randomly-isolated object $r \in$ *Conc*(SITE).

Random-Isolation Principle 2 (Proofs) *Given allocation site* SITE *and safety property* ϕ *over concrete objects, to establish that* $\forall o : Conc(\text{SITE}) . \phi(o)$ *holds, it is sufficient to establish that the augmented formula*

$\forall o : Conc(\textsc{Site}) \,.\, is_ri(o) \rightarrow \phi(o)$ holds, where the predicate is_ri is true for the randomly isolated object (allocated at site SITE*).*

Proof. The proof is by contradiction. Let Prog be a (concrete) concurrent program, $\mathsf{Prog}^\sharp$ be an (abstract) concurrent program obtained via the random-isolation abstraction, where the set of behaviors of $\mathsf{Prog}^\sharp$ over-approximates the set of behaviors of Prog, i.e., $\mathsf{Prog}^\sharp$ is a sound abstraction of Prog.

Consider the case where the abstract program analyzer—i.e., the software model checker—has verified that ϕ holds for the randomly-isolated object in $\mathsf{Prog}^\sharp$, and also that there exists a concrete trace where for some object $o_{\textsc{Site}} \in Conc(\textsc{Site})$, $\phi(o_{\textsc{Site}})$ does not hold. Because of random isolation, the randomly-isolated object $r_{\textsc{Site}}$ is just as likely to be $o_{\textsc{Site}}$ as it is to be any other concrete object. Thus, the model checker must consider the case that $r_{\textsc{Site}}$ is $o_{\textsc{Site}}$. Because the property holds for $r^\sharp_{\textsc{Site}}$, and because $r^\sharp_{\textsc{Site}}$ represents $o_{\textsc{Site}}$ in the trace under consideration, then the property must also hold for $o_{\textsc{Site}}$, which is a contradiction under the above listed assumptions. □

Random-Isolation Principle 2 is crucial for applying software model checking to the analysis of concurrent programs. In particular, it allows the model checker to consider only *finite*-data abstract programs yet still be able to prove safety properties of *infinite*-data concrete programs.[9] (§10 discusses related techiques for proving safety properties of infinite-data programs.)

***Random Isolation in* EMPIRE** Before describing the technical details of how random isolation is implemented, we highlight the benefits of random isolation as used in EMPIRE.

EMPIRE attempts to establish that all executions of a concurrent Java program Prog are AS-serializable with respect to the objects allocated at a user-specified allocation site SITE. Formally, let $\phi(o_{\textsc{Site}})$ be a formula stating that an execution is AS-serializable with respect to the concrete object $o_{\textsc{Site}}$. EMPIRE's goal is then to establish that for all executions,

$$\psi_1 = \forall x \in Conc(\textsc{Site}) \,.\, \phi(x).$$

Applying random-isolation abstraction allows us, via Random Isolation Principle 2, to change the goal to that of establishing the property:

$$\psi_2 = \forall x \in Conc(\textsc{Site}) \,.\, \mathtt{is_ri}(x) \rightarrow \phi(x).$$

Eliminating the implication, we rewrite ψ_2 as follows:

$$\psi_3 = \forall x : Conc(\textsc{Site}) \,.\, \neg\mathtt{is_ri}(x) \vee \phi(x).$$

Finally, to establish ψ_3 via a software model checker, we perform the following steps:

1. An abstract program $\mathsf{Prog}^\sharp$ is generated via the random-isolation abstraction.

[9] We say finite-data and not finite-state because each thread has an unbounded-sized stack, giving rise to an infinite-state space.

2. Formula ψ_3 is reinterpreted in the abstraction:
$$\psi_3^\sharp = \forall x^\sharp \in \{r^\sharp_{\textsc{Site}}, o^\sharp_{\textsc{Site}}\} \,.\, \neg\mathtt{is_ri}(x^\sharp) \vee \phi(x^\sharp).$$
The universal quantification is now over the two abstract objects defined by the abstract program.[10]
3. Model checker IPAMC is used to check if there exists an execution of the abstract program $\mathsf{Prog}^\sharp$ that satisfies the *negation* of the formula. If IPAMC reports that no such execution exists, then the original formula $\psi_3^\sharp$ holds. The negation is defined as:
$$\neg\psi_3^\sharp = \exists x^\sharp \in \{r^\sharp_{\textsc{Site}}, o^\sharp_{\textsc{Site}}\} \,.\, \mathtt{is_ri}(x^\sharp) \wedge \neg\phi(x^\sharp).$$
To use IPAMC to determine if an execution satisfies $\neg\psi_3^\sharp$, EMPIRE must actually make *fourteen* separate queries. The reason being that the negated formula, $\neg\phi(x^\sharp)$, is precisely defined by the fourteen problematic access patterns listed in Tab. 1.

The motivation for random isolation is to allow strong updates on the abstract lock of an abstract object, where a strong update corresponds to a definite lock acquire or definite lock release. Strong updates are enabled in EML by defining an EML lock for each non-summary object. Because of random isolation, the state of the Java lock that is associated with the randomly-isolated instance $r_{\textsc{Site}}$ can be modeled precisely by the state of the special abstract object $r^\sharp_{\textsc{Site}}$. That is, the acquiring and releasing of the lock for $r_{\textsc{Site}}$ by a thread of execution can be modeled by a strong update on the state of $r^\sharp_{\textsc{Site}}$, thus allowing the analyzer to disallow certain thread interleavings when performing state-space exploration on the generated EML program, and thereby improving the precision of the analysis.

In contrast, because sound tracking of the lock state for a summary object generally would result in the "possibly-held" state, EML programs have no locks for summary objects: their modeled behaviors are not restricted by synchronization primitives. This approach provides a sound, finite model of the locking behavior of $\mathsf{Prog}^\sharp$. (It is an over-approximation because the absence of locks on summary objects can cause the abstract program to have thread interleavings not found in the concrete Java program.)

6.1 Implementing Random Isolation

Random isolation can be implemented via a source-to-source transformation of the concrete Java program. For expository purposes, we break down the discussion of the source-to-source transformation into three separate rewriting steps and a step in which the abstract program is generated:

1. *Random Tagging:* The first transformation produces a residual program that for any execution of the original

[10] The concrete and abstract formulas, ψ_3 resp. $\psi_3^\sharp$, are related via the *Embedding Theorem* (Sagiv et al., 2002). The reader is referred to Harris et al. (2009, Lemma 1 and Theorem 1) for correctness proofs in the context of three-valued logic and universal quantification in abstractions of logical structures.

program Prog, at most one concrete object is randomly chosen to be tagged as special, i.e., at most one concrete object r is randomly isolated.
2. *Synchronized-Method Elimination:* The second transformation makes explicit the synchronization operation for a Java `synchronized` method. Recall that a `synchronized` method implicitly acquires the lock associated with the implicit `this` parameter. This transformation makes the synchronization explicit by erasing `synchronized` from method declarations and introducing their semantically equivalent `synchronized` blocks.
3. *Synchronized-Block Case Analysis:* Finally, synchronized blocks are the only source of lock and unlock operations. When analyzing the program model, the lock-state of a lock can generally only be determined precisely for randomly-isolated objects, i.e., those introduced by the first transformation above. However, due to aliasing it might not be possible to determine statically whether a reference uniquely refers to a randomly-isolated object or not. The final transformation introduces case-analysis expressions to determine whether an object reference `ref` in a Java expression "`synchronized(ref){...}`" solely refers to a randomly-isolated object or not.
4. *Defining the Abstract Program:* After the source-to-source transformation steps, an abstract program $\mathsf{Prog}^\sharp$ is generated from the transformed concrete Java program Prog.

We now discuss each transformation step in turn. To assist the reader, the transformation steps are illustrated for the example program shown in Listing 2.

Random Tagging: As explained above, the purpose of the first source-to-source transformation is to modify the program so that at most one concrete object is randomly tagged on any given run of a program. Because we do not wish to modify class definitions (e.g., by instrumenting each class to have an additional "`tag`" field), our approach to random tagging is to (i) generate a fresh allocation site; and (ii) ensure that at most one object can be allocated from the fresh allocation site for any given run of the program. Thus, random tagging corresponds to being allocated from the new allocation site. An additional benefit of this transformation is that it plays nicely with the allocation-site abstraction, the basis for the random-isolation abstraction. Specifically, by definition of the allocation-site abstraction, the introduction of a new allocation site in the program's source code causes the allocation-site abstraction to have an additional abstract object. In essence, our implementation of the random-isolation abstraction extends the allocation-site abstraction with special semantics to abstract objects that correspond to the newly introduced allocation sites—namely, the allocation-site abstraction is extended with the knowledge that the corresponding abstract objects are *non-summary* objects.

Concretely, let us consider the random-tagging transformation in the context of the example program in Listing 2, where the allocation site of interest is on line 22 and is repeated below for convenience.

$$\texttt{return}\ \texttt{new}_{\textsc{Site}}\ \texttt{Stack}(); \tag{1}$$

The Random-Tagging transformation involves transforming the $\texttt{new}_{\textsc{Site}}$ statement into

$$\begin{aligned}(\texttt{rand}()\ \&\&\ \mathcal{F}_{\textsc{SiteRi}}.\texttt{compareAndSet}(\texttt{false},\texttt{true}))\\ ?\ \texttt{new}_{\textsc{SiteRi}}\ \texttt{Stack}()\ :\ \texttt{new}_{\textsc{Site}}\ \texttt{Stack}();\end{aligned} \tag{2}$$

Site SITE from code fragment (1) is transformed into a conditional-allocation site, where the conditional performs a compareAndSet of a newly introduced global flag $\mathcal{F}_{\textsc{SiteRi}}$.[11] The global flag $\mathcal{F}_{\textsc{SiteRi}}$ can be set to true at most once, thus ensuring that at most one concrete object $r_{\textsc{Site}}$ can ever be allocated at the generated site SITERI. That is, $\mathcal{F}_{\textsc{SiteRi}}$ guarantees that $|Conc(\textsc{SiteRi})| \leq 1$ for all possible executions of the program; consequently, the size of the set that is the concretization of $r^\sharp_{\textsc{Site}}$ must also be less than or equal to one.

The observant reader might be concerned that an AS-serializability violation could occur on an object allocated from SITE that is not tagged. For example, an execution that contains an AS-serializability violation but never allocates an object from the fresh allocation site SITERI would satisfy this criteria. Fortunately, this is not a problem because our analysis considers all executions of the abstract program. The following corollary makes formal why such executions—those where the fresh allocation site is never executed—need not be considered:

Corollary 1. *For performing AS-serializability-violation detection, we only have to be concerned with executions in which $r_{\textsc{Site}}$ is eventually allocated.*

Proof. The claimed property is a consequence of the use of randomness in code fragment 2. If there were a trace in which $r_{\textsc{Site}}$ was not allocated but the trace did in fact contain an AS-serializability violation, then because of the use of randomness, there must exist a similar trace in which the $r_{\textsc{Site}}$ object was allocated and the violation occurred on that object. That is, the need to only consider traces in which $r_{\textsc{Site}}$ is (eventually) allocated is a corollary of *Random-Isolation Principle 2*. All such traces have the property that the global flag $\mathcal{F}_{\textsc{SiteRi}}$ must eventually be set to true (see $\neg\psi^\sharp_3$ in item 3 on page 10). □

Synchronized-Method Elimination: We now discuss the second transformation, which makes explicit all lock acquisitions and releases in the Java program Prog. It is desirable to have explicit locking operations because the analysis requires the ability to distinguish between locking operations performed on randomly tagged objects—i.e., concrete objects whose abstract counterpart is guaranteed to be a non-summary object—versus all other objects. Making the locking operations explicit allows uniform treatment of the case analysis, which is the subject of the third transformation. Because this transformation is trivial, we describe it by example.

---

[11] `b.compareAndSet(expect,update)` is supported by class `java.util.concurrent.atomic.AtomicBoolean`. It atomically sets Boolean value `b` to `update` if the current value of `b` is the expected value `expect`. It is important to use an *atomic* operation in the test in (2); if $\mathcal{F}_{\textsc{SiteRi}}$ were not set atomically, the source-to-source transformation would introduce a race condition that would allow multiple objects to be allocated at the newly introduced allocation site SITERI.

Consider the method $\texttt{Stack.size}()$ in Listing 2, repeated below for convenience:

$$@\texttt{atomic sync int size}()\ \{\ \texttt{return count} + 1;\ \}$$

The lock associated with the $\texttt{this}$ object is implicitly acquired when $\texttt{size}()$ is invoked, and released upon return. To make the acquisition and release explicit, the method is transformed as follows:

$$@\texttt{atomic int size}()\ \{\ \texttt{sync(this)}\ \{\ \texttt{return count} + 1;\ \}\}$$

Observe that the synchronized keyword has been removed from the method declaration, and the entire body of the method has been enclosed in a synchronized block that acquires the lock associated with the (implicit) $\texttt{this}$ parameter. (For static synchronized methods, i.e., for synchronized *class* methods, the $\texttt{this}$ parameter to the synchronized block is replaced with the name of the Java class in which the method declaration occurs.)

Synchronized-Block Case Analysis: The last transformation performs a case analysis on the points-to information associated with the reference parameter to a synchronized block. Consider the body of the transformed $\texttt{size}$ method just described:

$$\texttt{sync(this)}\ \{\ \texttt{return count} + 1;\ \}$$

in the context of the program with a transformed allocation site as described in the *Random-Tagging* transformation.

$$(\texttt{rand}()\ \&\&\ \texttt{testAndSet}(\mathcal{F}_{\text{SITERI}}))$$
$$?\ \texttt{new}_{\text{SITERI}}\ \texttt{Stack}()\ :\ \texttt{new}_{\text{SITE}}\ \texttt{Stack}();$$

The $\texttt{size}$ method could be invoked on either the randomly-tagged object r_{SITERI} or an object allocated from the original allocation site o_{SITE}. When analyzing the program under random-isolation semantics, the analysis can generally only prove that the abstract object $r^\sharp_{\text{SITERI}}$ corresponding to the randomly-tagged object r_{SITERI} is a non-summary object. Thus, locking operations, which constitute strong updates to the abstract state of a lock object, can generally only be performed on the object referred to by a Java reference if the analysis can prove that the reference only refers to a randomly-tagged object. In points-to analysis terms, where the set of abstract objects that a reference (e.g, $\texttt{this}$) may point to is denoted by $\mathsf{Pts}(\texttt{this})$, acquiring and releasing a lock is generally only possible when the analysis can prove that $\mathsf{Pts}(\texttt{this}) = \{r^\sharp_{\text{SITERI}}\}$.

Our final transformation embeds a case analysis that enables such a distinction to be made. The key observation is that the desired case—that $\mathsf{Pts}(\texttt{this}) = \{r^\sharp_{\text{SITERI}}\}$—can be precisely checked for dynamically. Specifically, the dynamic check tests whether or not the reference refers to the randomly-tagged (concrete) object r_{SITERI}. We call the checking method $\texttt{is_ri}$ because it is a test whether a reference *is* a reference to a *randomly isolated* object. The concrete semantics of $\texttt{is_ri}$ is as follows:

$$\texttt{is_ri(t)} =_{\text{def}} \quad \mathcal{F}_{\text{SITERI}} \wedge t = r_{\text{SITERI}}$$

The final transformation for a synchronized block on reference t is as follows:

$$//\texttt{is_ri(t)} = \mathit{true}\ \textit{implies}\ \mathsf{Pts}(t) = \{r_{\text{SITERI}}\}$$
$$\texttt{if (is_ri(t))}\ \{\ \texttt{sync(t)}\ \{\ \texttt{body}\ \}\ \}$$
$$//\texttt{is_ri(t)} = \mathit{false}\ \textit{implies}\ \mathsf{Pts}(t) \not\ni r_{\text{SITERI}}$$
$$\texttt{else}\ \{\ \texttt{sync(t)}\ \{\ \texttt{body}\ \}\ \}$$

Observe that the true and false branches of the $\texttt{is_ri}$ test perform the same concrete actions. This is crucial because the transformation must not alter the concrete semantics of the program. That is, one should view $\texttt{is_ri}$ as performing a metadata test, where the metadata of interest is whether or not the referred to object has been randomly tagged. During EML generation (discussed in §7), the metadata test determines whether or not the generated EML program will contain an action that changes the state of an abstract lock.

During points-to analysis, the interpretation of the call on $\texttt{is_ri}$ performs a case analysis on $\mathsf{Pts}(\texttt{t})$. Specifically, the abstract interpretation of $\texttt{is_ri}$ performs the abstract test "$\mathcal{F}_{\text{SITERI}} \wedge \texttt{t} = r^\sharp_{\text{SITERI}}$", which is just the corresponding abstract semantics of the concrete semantics of $\texttt{is_ri}$ given above. We extend the points-to analysis to interpret the $\texttt{is_ri}$ method call as follows: when following the true branch of the condition, the points-to analysis performs an "$\texttt{assume}\ \mathsf{Pts}(\texttt{t}) = \{r^\sharp_{\text{SITERI}}\}$"; and when following the false branch of the condition, the points-to analysis performs an "$\texttt{assume}\ \mathsf{Pts}(\texttt{t}) = \mathsf{Pts}(\texttt{t}) \setminus \{r^\sharp_{\text{SITERI}}\}$".

Remark 2. The final transformation can be avoided, i.e., the transformation that implements a case analysis via $\texttt{is_ri}$, when the analysis can already prove that $\mathsf{Pts}(t) = \{r^\sharp_{\text{SITERI}}\}$. Such a property can often be established by means of an object-sensitive interprocedural control-flow graph (ICFG). For readers familiar with object-sensitive analysis in the style of Milanova et al. (2005), and its corresponding object-sensitive call graph, the technique we use is similar to Milanova et al. (2005). Briefly, an object-sensitive call graph CG models the interprocedural control flow of a program, qualified by the identities of the receiver objects of methods: there is a node in CG for each method of the program for each context—i.e., each abstract object that can act as the receiver of the invocation—in which it can be invoked (Milanova et al., 2005). An object-sensitive points-to analysis associates points-to facts with the nodes of CG, thus computing different points-to facts for different object contexts of the same method. Because these two analysis artifacts are object-sensitive, their respective dataflow facts make a distinction between objects allocated at SITERI and objects allocated at SITE. For example, referring back to the program in Listing 2, there would be two copies of the control-flow graph (CFG) for the method $\texttt{Stack.size}$, one for object context $o^\sharp_{\text{SITE}}$ and one for object context $r^\sharp_{\text{SITERI}}$. Thus, inside of the CFG for $\texttt{Stack.size}$ with object context $r^\sharp_{\text{SITERI}}$, an analysis is able to take advantage of the fact that the special Java $\texttt{this}$ variable refers solely to the non-summary object $r^\sharp_{\text{SITERI}}$.

Defining the Abstract Program. After the source-to-source transformations have been performed, the resulting program is ready for abstraction; the corresponding abstract program $\mathsf{Prog}^\sharp$ will operate over a set of abstract objects consisting of $o^\sharp_{\text{SITE}}$ for each allocation site SITE in the program (including the special abstract object $r^\sharp_{\text{SITERI}}$ for the generated allocation site SITERI). Because threads in Java are objects themselves, $\mathsf{Prog}^\sharp$ now has a finite number of abstract threads, where each thread is associated with an allocation site. (We can, of course, generate multiple copies of each thread as needed. That is, the number of threads is actually a parameter of $\mathsf{Prog}^\sharp$.)

6.2 Summary

We conclude with three points about random isolation and its implementation via source-to-source transformations:

1. Observe that if the randomly-isolated abstract object $r^\sharp_{\text{SITERI}}$ is folded into the summary object $o^\sharp_{\text{SITE}}$, then the abstraction reduces to the allocation-site abstraction.
2. From the above observation, we can conclude that random isolation can be used to improve the precision of any analysis that must reason about an unbounded set of indistinguishable objects, especially analyses that already use the allocation-site abstraction. Moreover, random isolation is particularly effective when an analysis needs the ability to perform strong updates on the abstract state of an object.
3. Even though the transformations have been presented in the context of AS-serializability-violation detection, random isolation is a generic approach that can be applied wherever an analysis needs to distinguish between $r^\sharp_{\text{SITERI}}$ and $o^\sharp_{\text{SITE}}$ to perform a strong update. In particular, the splitting of allocation sites to distinguish a random individual and the injection of case-analysis expressions to isolate the actions performed on that individual has applications outside of AS-serializability violation detection.

7 EML Generation

Once the abstract program $\mathsf{Prog}^\sharp$ has been generated from a concurrent Java program Prog, the next step in EMPIRE is to generate an EML program EProg. We now discuss the EML-generation process.

To model the randomly-isolated abstract object $r^\sharp_{\text{SITE}}$, EProg defines a shared memory location m_f for each field f of the class T, and also an EML lock l_{SITERI} to model the lock associated with $r^\sharp_{\text{SITE}}$. The status of the global flag $\mathcal{F}_{\text{SITERI}}$ is modeled by the EML lock l_{SITERI} being allocated or not. As shown in §6.1 (see Corollary 1), we are only concerned with executions in which l_{SITERI} is eventually allocated. Hence we need only be concerned with traces of the EML program in which "$\mathsf{alloc}\ l_{\text{SITERI}}$" appears somewhere.

Let $\mathsf{Threads}$ be the set of all subclasses of `java.lang.Thread`. For each $\theta \in \mathsf{Threads}$, and for each allocation site SITE_θ that allocates an instance of θ, EProg defines an EML process $\mathsf{Proc}_{\text{SITE}_\theta}$ that models the behavior of one instance of θ that is allocated at SITE_θ. Also, EProg defines an EML process $\mathsf{Proc}_{\mathsf{main}}$ that models the Java thread that begins execution of the `main` method.

The functions of an EML process Proc correspond one-to-one with the methods of the Java program, i.e., for each Java method m there is an EML function f_m. The labeled-flow graph $\mathcal{G}_{f_m}$ is defined from the control-flow graph CFG_m that is associated with m.[12] The control-flow graph CFG_m consists of a set of Java statements Stmts, a successor relation $\mathsf{Succ} \subseteq \mathsf{Stmts} \times \mathsf{Stmts}$, and has distinct entry and exit statements. The translation from CFG_m to $\mathcal{G}_{f_m}$ is straightforward. There is a node $n_{\mathsf{stmt}} \in \mathsf{Nodes}_{f_m}$ for each $\mathsf{stmt} \in \mathsf{Stmts}$. There is a labeled edge $(n_{\mathsf{stmt}}, a_{\mathsf{stmt}}, n_{\mathsf{stmt}'})$ for each pair of statements $(\mathsf{stmt}, \mathsf{stmt}') \in \mathsf{Succ}$. The label a_{stmt} models the execution of the Java statement stmt. Tab. 3 shows the label a_{stmt} that is generated for a Java statement stmt. Note that (i) `synchronized` blocks have been compiled down to the lower-level Java bytecode statements `monitorenter` and `monitorexit`; (ii) label generation must know if a method is a unit of work; and (iii) a read or write access to the contents of array field f of T is treated as a read or write access to the field f.[13]

8 Verifying AS-serializability of EML Programs

An EML program has a set of shared-memory locations, S_{Mem}, a set of EML locks, S_{Locks}, and a set of EML processes, S_{Procs}. To verify AS-serializability of an EML program, EMPIRE generates a number of queries that are then answered by IPAMC (Kidd et al., 2009b, 2011)—a model checker for multi-PDSs with reentrant locks, whose decision procedure can answer the types of queries required by EMPIRE.

Remark 3. Even though many analyses for multi-PDSs are undecidable, Kahlon and Gupta (2007) showed that property checking for certain fragments of temporal logic is decidable. For a detailed account of the decision procedure of Kidd et al. (2011) and the characteristics of the problem that render it decidable, the reader is referred to Kidd (2009).

A query is generated for each pair $(m, m') \in S_{\mathsf{Mem}} \times S_{\mathsf{Mem}}$ for the fourteen interleaving scenarios. Pairs are used because the interleaving scenarios are defined in terms of at most two locations from an atomic set (cf. Tab. 1). Moreover, AS-serializability-violation detection is asymmetric in that it is performed with respect to an individual EML process Proc. In total, EMPIRE generates $O(|S_{\mathsf{Procs}}| * 14 * (|S_{\mathsf{Mem}}|^2))$ queries for an EML program.

A generated query Π consists of a (i) *pushdown system* (PDS) for each EML process (§8.2), and (ii) an *indexed phase*

[12] When using an object-sensitive analysis as discussed in Remark 2, there will be multiple labeled-flow graphs for a Java method m, one for each object context in which it could be invoked.

[13] Our decision to treat an array and its contents as one field is not strictly necessary. Using a more-refined analysis that can reason about arrays (Sagiv et al., 2002; Gulwani et al., 2008; Dillig et al., 2010), a more precise model could be generated.

stmt	a_{stmt}	Condition
o.m()	call m	
entry	unitbegin	this= $r^\sharp_{\text{SITE}}$, CFG$_m$ is a unit of work
exit	unitend	this= $r^\sharp_{\text{SITE}}$, CFG$_m$ is a unit of work
x = o.f, x = o.f[i]	read m_f	$r^\sharp_{\text{SITE}} \in \mathsf{Pts}(o)$
o.f = x, o.f[i] = x	write m_f	$r^\sharp_{\text{SITE}} \in \mathsf{Pts}(o)$
new$_{\text{SITERI}}$ T	alloc $l_{r^\sharp_{\text{SITE}}}$	
monitorenter o	lock $l_{r^\sharp_{\text{SITE}}}$	$\mathsf{Pts}(o) = \{r^\sharp_{\text{SITE}}\}$
monitorexit o	unlock $l_{r^\sharp_{\text{SITE}}}$	$\mathsf{Pts}(o) = \{r^\sharp_{\text{SITE}}\}$
o.start()	start Proc$_{\text{SITE}_\theta}$	$o^\sharp_{\text{SITE}_\theta} \in \mathsf{Pts}(o)$.
*	skip	

Table 3. Java statement types for CFG$_m$, their corresponding EML labels, and the condition necessary to generate the EML label. The final row is a catchall for the Java statements that are either not modeled in EML or that do not satisfy the listed condition.

Rule	Control flow modeled
$\langle p, n_1\rangle \xhookrightarrow{s_1} \langle p, n_2\rangle$	Intraprocedural edge $n_1 \to n_2$
$\langle p, n_c\rangle \xhookrightarrow{s_c} \langle p, e_f\ r_c\rangle$	Call to f, with entry e_f, from n_c that returns to r_c
$\langle p, x_f\rangle \xhookrightarrow{s_f} \langle p, \epsilon\rangle$	Return from f at exit x_f

Table 4. The encoding of a call graph's and CFG's edges as PDS rules. The action a denotes the abstract behavior of executing that edge.

automaton (IPA)—a restricted kind of non-deterministic finite automaton (NFA)—that recognizes EML execution traces containing an AS-serializability violation involving one or two memory locations (§8.4). The remainder of this section proceeds as follows: §8.1 formally defines PDSs, multi-PDSs, and IPAs; §8.2 defines the translation from an EML process to a PDS; §8.3 describes how IPAMC handles reentrant locks; and §8.4 defines the IPA for an atomic-set/problematic-access-pattern pair.

8.1 Definitions

A pushdown system (PDS) naturally models the interprocedural control flow of an EML process (see Tab. 4). In essence, a PDS is an NFA equipped with a stack. The control states of the NFA portion of a PDS models the global state of an EML process, while the stack is used to ensure that only interprocedurally valid paths are modeled.

Definition 1. A *pushdown system* (PDS) is a tuple $\mathcal{P} = (P, \Gamma, \mathsf{Lab}, \Delta, c_0)$, where P is a finite set of control locations; Γ is a finite set of stack symbols; Lab is a finite set of labels (or actions); $\Delta \subseteq (P \times \Gamma) \times \mathsf{Lab} \times (P \times \Gamma^*)$ is a finite set of labeled-transition rules, where a rule is denoted by $r = \langle p, \gamma\rangle \xhookrightarrow{a} \langle p', u'\rangle$; and $c_0 = \langle p_0, \gamma_0\rangle$ is the initial configuration of $\mathcal{P}$. A *configuration* c of $\mathcal{P}$ is a pair $\langle p \in P, u \in \Gamma^*\rangle$. For a rule $r = \langle p, \gamma\rangle \xhookrightarrow{a} \langle p', u'\rangle$, we use $\mathsf{lab}(r)$ to denote r's label a. Without loss of generality, we restrict there to be at most two right-hand-side stack symbols of a rule. A rule with zero, one, and two right-hand-side stack symbols is called a *pop*, *step*, and *push* rule, respectively.

A concurrent program that synchronizes via locks, i.e., an EML program, is modeled by a *multi-PDS* . Informally, a multi-PDS consists of a finite set of PDSs and a finite set of locks. The intention is that each PDS models a thread, and that the PDSs acquire and release locks to perform global synchronization. We assume that locks are acquired and released in a well-nested fashion—locks are released in the opposite order in which they are acquired—and in synchrony with a PDS's push and pop rules. In fact, the latter assumption is all that is necessary as it implies the former.

Definition 2. A *multi-PDS* consists of a finite number of PDSs $\mathcal{P}_1, \ldots, \mathcal{P}_n$, where each PDS $\mathcal{P}_j = (P_j, \Gamma_j, \mathsf{Lab}_j, \Delta_j, c_0^j)$, that synchronize via a finite set of locks $S_{\mathsf{Locks}} = \{l_1, \ldots, l_{|S_{\mathsf{Locks}}|}\}$. The actions Lab of each PDS consist of lock-acquires ("$(_i$") and releases ("$)_i$") for $1 \leq i \leq |S_{\mathsf{Locks}}|$, plus symbols from Σ, a finite alphabet of non-parenthesis symbols. A ***global configuration*** $(c_1, \ldots, c_n, \bar{o})$ is a tuple consisting of:

- a local configuration c_j for each PDS $\mathcal{P}_i$, $1 \leq j \leq n$; and
- an ***ownership array*** $\bar{o}$ of length $|S_{\mathsf{Locks}}|$, in which each entry indicates the owner of a given lock: for each $1 \leq i \leq |S_{\mathsf{Locks}}|$, $\bar{o}[i] \in (\{\bot, 1, \ldots, n\} \times \mathcal{N})$ is a pair where the first component indicates the identity j of the PDS $\mathcal{P}_j$ that holds lock l_i ($\bot$ signifies that l_i is currently not held by any PDS), and the second component is a non-negative number that indicates the number of times that a PDS has (re)acquired a lock.

The ***initial global configuration*** $g_0 = (c_0^1, \ldots, c_0^n, \bar{o}_0)$, where c_0^i is the initial configuration of PDS $\mathcal{P}_i$, $1 \leq i \leq n$, and $\bar{o}_0$ is the initial ownership array that maps each entry $\bar{o}[i]$, $1 \leq i \leq |S_{\mathsf{Locks}}|$, to the ownership pair $(\bot, 0)$. For an ownership array $\bar{o}$, an update at position i for lock l_i to a new ownership pair p is denoted by $\bar{o}[i \mapsto p]$.

Definition 3. For multi-PDS $\Pi = (\mathcal{P}_1, \ldots, \mathcal{P}_n, S_{\mathsf{Locks}}, \Sigma)$, the *Reentrant Semantics* allows for a lock $l \in S_{\mathsf{Locks}}$ to be

reacquired by the PDS that owns the lock. In particular, two global configurations g and g' are in the relation $\rightsquigarrow$, denoted by $g \rightsquigarrow g'$, iff $g = (c_1, \ldots, c_j, \ldots, c_n, \bar{o})$ and one of the following holds:

1. $c_j \xrightarrow{a} c'_j$, $a \in \Sigma$, and $g' = (c_1 \ldots, c'_j, \ldots, c_n, \bar{o})$.
2. $c_j \xrightarrow{(_i} c'_j$, $\bar{o}[i] = (\bot, 0)$, and $g' = (c_1, \ldots, c'_j, \ldots, c_n, \bar{o}[i \mapsto (j, 1)])$.
3. $c_j \xrightarrow{(_i} c'_j$, $\bar{o}[i] = (j, z)$, and $g' = (c_1, \ldots, c'_j, \ldots, c_n, \bar{o}[i \mapsto (j, z + 1)])$.
4. $c_j \xrightarrow{)_i} c'_j$, $\bar{o}[i] = (j, z)$, $z > 1$, and $g' = (c_1, \ldots, c'_j, \ldots, c_n, \bar{o}[i \mapsto (j, z - 1)])$.
5. $c_j \xrightarrow{)_i} c'_j$, $\bar{o}[i] = (j, 1)$, and $g' = (c_1, \ldots, c'_j, \ldots, c_n, \bar{o}[i \mapsto (\bot, 0)])$.

The reflexive transitive closure of $\rightsquigarrow$ is denoted by $g \rightsquigarrow^* g'$. A multi-PDS execution π is a finite sequence steps $g_0 \rightsquigarrow g_1 \rightsquigarrow \ldots \rightsquigarrow g_k$. The word w assocated with π is the sequence of actions from Σ—locking operations are not exposed—that correspond to each transition step by item 1 above.

Given a multi-PDS, IPAMC is a model checker that determines whether there exists an interleaved execution π of a multi-PDS such that the word w associated with π is recognized by an *indexed phase automaton* (IPA).

Definition 4. An ***indexed phase automaton*** (IPA) is a tuple (Q, Id, Σ, δ), where Q is a finite, totally ordered set of states $\{q_1, \ldots, q_{|Q|}\}$, Id is a finite set of thread identifiers, Σ is a finite alphabet, and $\delta \subseteq Q \times Id \times \Sigma \times Q$ is a transition relation. The transition relation δ is restricted to respect the order on states: for each transition $(q_x, i, a, q_y) \in \delta$, either $y = x$ or $y = x + 1$. We call a transition of the form (q_x, i, a, q_{x+1}) a ***phase transition***. The initial state is q_1, and the final state is $q_{|Q|}$.

The restriction on δ in Defn. 4 ensures that the only loops in an IPA are self-loops on states, which is crucial for keeping the checking problem decidable (Kidd, 2009). We assume that for every x, $1 \leq x < |Q|$, there is only one phase transition of the form $(q_x, i, a, q_{x+1}) \in \delta$. (An IPA that has multiple such transitions can be factored into a set of IPAs, each of which satisfy this property.) Finally, we only consider IPAs that recognize a non-empty language, which means that an IPA must have exactly $(|Q| - 1)$ phase transitions, i.e., any accepting run of an IPA will contain only a bounded number of phase transitions.

8.2 Modeling an EML *Process*

We now discuss the generation of a PDS for an EML process Proc. For expository purposes, we break the translation into two steps.

- *Step 1:* a PDS $\mathcal{P}_1$ is generated that models the interprocedural control flow of Proc. In this stage, all locking operations—i.e., inter-process communication—are not modeled by $\mathcal{P}_1$.
- *Step 2:* from PDS $\mathcal{P}_1$ a PDS $\mathcal{P}_2$ is generated that accounts for locking operations and thus implements the necessary interprocess communication that restricts the set of allowable schedules.

Step 1: The first step is straightforward: a single-state PDS $\mathcal{P}_1 = (P_1 = \{p\}, \mathsf{Lab}_1, \Gamma_1, \Delta_1, \langle p, e_{\mathsf{main}} \rangle)$ is generated using the rule templates depicted in Tab. 4, with Lab_1 being the set of all distinct EML statements used by Proc prefixed with the EML process's name. For example, if the EML statement is "read m" and the EML process's name is Proc, then Lab_1 will contain "Proc.read m". Including the EML process's name in the PDS action enables the violation monitor and locks to know which EML process performs an action (cf. the PDS rules for an EML lock above). $\mathcal{P}_1$ captures the interprocedural control flow of Proc. There is one exception, the EML statement "start $\mathsf{Proc}_{\mathrm{SITE}_\theta}$" does not include Proc as a prefix because a thread can be started by any other thread (but only started one time).

Step 2: PDS $\mathcal{P}_2$ is PDS $\mathcal{P}_1$ augmented to account for lock allocation. (Recall that a lock must be allocated before it can be used.) The set of control locations P_1 of $\mathcal{P}_1$ is expanded to include a boolean flag for each lock l. If the flag is true then l has been allocated, otherwise an EML process has yet to allocate l.

From Proc's point of view, there are two ways that a lock can be allocated: either Proc allocates a lock or another EML process Proc′ allocates the lock. If Proc allocates the lock l, then there will be a PDS rule of the form $\langle p, \gamma \rangle \xhookrightarrow{\mathsf{Proc}.alloc\ l} \langle p', u \rangle$. The corresponding rule in PDS $\mathcal{P}_2$'s rule set Δ_2 must ensure that the control location on the left-hand-side has the flag for l set to false, and the control location on the right-hand-side has the flag for l set to true. Otherwise Proc′ allocates l. In this case, Proc has no way of knowing when Proc′ allocates l, and therefore must *guess* when the allocation occurred. Guessing is modeled by non-deterministically invoking the guess method, which simply guesses that another EML process allocates l. Because of non-determinism, the guess method causes Proc to consider all possibilities of lock allocation (i.e., the cross product of S_{Procs} and S_{Locks}).

Formally, PDS $\mathcal{P}_2 = (P_2, \mathsf{Lab}_2, \Gamma_2, \Delta_2, \langle \emptyset, e_{\mathrm{main}} \rangle)$, where

- $P_2 = 2^{S_{\mathsf{Locks}}}$: a control location is a set of flags s denoting which locks have been allocated. The set of control locations P_1 is not used because it is the singleton set $\{p\}$.
- $\mathsf{Lab}_2 = \mathsf{Lab}_1 \cup \{P'.\mathsf{alloc}\ l \mid P' \in (S_{\mathsf{Procs}} \setminus \{\mathsf{Proc}\}), l \in S_{\mathsf{Locks}}\}$: Lab_2 is Lab_1 augmented to include actions that allow for $\mathcal{P}_2$ to guess when another EML process Proc′ allocates a lock.
- $\Gamma_2 = \Gamma_1 \cup \{\mathsf{guess}\}$: Γ_2 includes the stack symbol guess that implements the guessing procedure.
- Δ_2 is defined from Δ_1 as shown in Tab. 5. Row 2 ensures that no lock is allocated more than once; row 3 ensures that a lock is not used before being allocated; and rows 4 and 5 ensure that the shared-memory locations are not accessed

Action a	Rule $\langle p, \gamma \rangle \xhookrightarrow{a} \langle p, w \rangle \in \Delta_1$	
τ, start $\mathcal{P}'$	$\{ \langle s, \gamma \rangle \xhookrightarrow{a} \langle s, w \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}}$ $\}$
alloc l	$\{ \langle s, \gamma \rangle \xhookrightarrow{P.a} \langle s \cup \{l\}, w \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge l \notin s$ $\}$
lock/unlock l	$\{ \langle s, \gamma \rangle \xhookrightarrow{P.a} \langle s, w \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge l \in s$ $\}$
read/write m	$\{ \langle s, \gamma \rangle \xhookrightarrow{P.a} \langle s, w \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge l_{\text{SITER1}} \in s$ $\}$
ubegin/uend	$\{ \langle s, \gamma \rangle \xhookrightarrow{P.a} \langle s, w \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge l_{\text{SITER1}} \in s$ $\}$
$\star$	$\{ \langle s, \gamma \rangle \xhookrightarrow{\tau} \langle s, \mathsf{guess}\ \gamma \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}}$ $\}$
$\star$	$\{ \langle s, \mathsf{guess} \rangle \xhookrightarrow{P'.\mathsf{alloc}\ l} \langle s \cup \{l\}, \epsilon \rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge l \notin s \wedge P' \in (S_{\mathsf{Procs}} \setminus \{P\})$ $\}$

Table 5. Each row defines a set of PDS rules in Δ_2 from a rule in Δ_1. The control location p from a rule in Δ_1 is not repeated because all rules in Δ_1 are single-control-location rules. The condition for generating a rule reflects that certain actions can only occur when a lock has been allocated, e.g., acquiring a lock l can only occur after l has been allocated (see §8.2).

before $r^{\sharp}_{\text{SITE}}$ has been allocated. Row 6 defines rules that invoke the "guessing" procedure for each configuration of $\mathcal{P}_2$. Guessing is necessary because an EML process cannot know when another EML process allocates a lock. Row 7 defines rules that implement the guessing procedure: from control location s, $s \subseteq S_{\mathsf{Locks}}$, guess that EML process $P' \in (S_{\mathsf{Procs}} \setminus \{P\})$ allocates a lock $l \in (S_{\mathsf{Locks}} \setminus s)$, and return back to the caller in the control location $s \cup \{l\}$. The guessing rule is then labeled with action $P'.\mathsf{alloc}\ l$.

The multi-PDS that models an EML program then has, for each EML process, a PDS $\mathcal{P}$ defined as just described, and the set of locks is merely the set of locks S_{Locks} of the EML program.

8.3 *Modeling an* EML *Lock*

The translation from an EML lock to a multi-PDS lock that is used by IPAMC is straightforward: there is a one-to-one correspondence between EML locks and the set of multi-PDS locks. The use of the notation S_{Locks} to represent both the set of EML locks and multi-PDS locks emphasizes this correspondence.

We next briefly describe how IPAMC replaces reentrant locks with non-reentrant locks via a transformation called *language-strength reduction* (Kidd et al., 2008). The transformation relies on the fact that lock acquisitions and releases are synchronized with a PDS's stack. Briefly, the technique involves transforming a PDS so that it pushes a special marker onto the stack the *first* time a lock is acquired, and also records the fact that the PDS acquired the lock in the PDS's state space. All subsequent lock acquisitions and and their matching releases do not change the state of the PDS. When the special marker is seen again on the stack, the PDS is about to execute the matching release for the initial lock acquisition, and thus actually release the lock. In essence, for PDS $\mathcal{P} = (P, \mathsf{Lab}, \Gamma, \Delta, c_0)$, the transformation defines a new PDS $\mathcal{P}' = (P', \mathsf{Lab}, \Gamma', \Delta', c_0')$, where

- $P' = P \times 2^{S_{\mathsf{Locks}}}$, where a member (p, s) of P' includes the original control state p of P and a set s that records the set of locks that $\mathcal{P}'$ currently holds.
- $\Gamma' = \Gamma \cup (\Gamma \times S_{\mathsf{Locks}})$ consists of the original stack alphabet Γ, and, in addition, a new set of symbols that enables $\mathcal{P}'$ to record on the stack the first time that a lock $l \in S_{\mathsf{Locks}}$ has been acquired.
- Δ' contains, for each rule $r \in \Delta$, a set of rules that maintain and update the set of held locks s for a control state (p, s), and record on the stack via a symbol (γ, l) that a lock has been acquired for the first time. The exact definition of Δ' is beyond the scope of this paper, and we refer the reader to Kidd et al. (2008) for further details.
- $c_0' = \langle (p_0, \emptyset), \gamma_0 \rangle$ is the initial configuration paired with $\emptyset$ to indicate that $\mathcal{P}'$ does not hold any locks.

After transforming $\mathcal{P}$ to be $\mathcal{P}'$, all nested lock acquisitions and releases have been removed. Hence, we can refine the Reentrant Semantics to disallow configurations where an ownership array $\bar{o}$ contains at position i for lock l_i an ownership pair (j, z) such that $z > 1$, which gives the *Non-Reentrant Semantics*. To distinguish between the Non-Reentrant and Reentrant Semantics, we will use j and $\bot$ to denote the ownership pair $(j, 1)$ and $(\bot, 0)$, respectively. The shorthand is sound because there can be no ambiguity, i.e., (j, z) where $z > 1$ is not allowed. Moreover, to distinguish the Non-Reentrant Semantics, we will use $\rightarrow$ instead of $\rightsquigarrow$ to denote the transition relation between global configurations.

Definition 5. The *Non-Reentrant Semantics* of a multi-PDS CP is defined by a transition relation $\rightarrow$ on global configurations. Two global configurations g and g' are in $\rightarrow$, denoted by $g \rightarrow g'$, iff $g = (c_1, \ldots, c_j, \ldots, c_n, \bar{o})$ and one of the following holds:

1. $c_j \xrightarrow{a} c_j'$, $a \notin \{ (_i,)_i \}$, and $g' = (c_1 \ldots, c_j', \ldots, c_n, \bar{o})$.
2. $c_j \xrightarrow{(_i} c_j'$, $\bar{o}[i] = \bot$, and $g' = (c_1, \ldots, c_j', \ldots, c_n, \bar{o}[i \mapsto j])$.
3. $c_j \xrightarrow{)_i} c_j'$, $\bar{o}[i] = j$, and $g' = (c_1, \ldots, c_j', \ldots, c_n, \bar{o}[i \mapsto \bot])$.

Note that items 1,2, and 3 correspond to items 1,2, and 5 of the Reentrant Semantics. The word w that for an execution π is defined the same as in the Reentrant Semantics.

8.4 Violation Monitor

The IPA $\mathcal{A}_{\mathsf{mon}}$ acts as a *violation monitor* that detects when one of the problematic access patterns occurs during a unit of work for a specific EML process Proc. To do so, it must track (i) the reads and writes to the shared-memory locations S_{Mem} by each EML process, and (ii) whether or not the target EML process Proc is executing a unit of work.

Tracking the reads and writes of EML processes requires only a finite amount of state, i.e., state to track which reads and writes of interest have been seen. Recall that units of work are reentrant because unit-of-work methods may be recursive or invoke other unit-of-work methods. Thus, tracking the unit-of-work status of Proc requires an infinite amount of state. Fortunately, the stack of the PDS for Proc is infinite-state, and the *language-strength-reduction* transformation (Kidd et al., 2008) can be used to fuse the unit-of-work depth count with the PDS stack. The transformation is analogous to the one discussed in §8.3. After applying the transformation, tracking the unit-of-work status for Proc requires only a finite amount of state, namely a Boolean flag, and thus tracking both the reads and writes of interest and the unit-of-work status of Proc can be performed by an IPA $\mathcal{A}_{\mathsf{mon}}$.

The IPA $\mathcal{A}_{\mathsf{mon}}$ accepts interleaved executions (traces) that contain the memory accesses specified by the problematic access pattern of interest. To make the discussion more concrete, we focus on the IPA $\mathcal{A}_{12}$ shown in Fig. 3, which tracks the problematic access pattern "$R_1(c); W_2(d); W_2(c); R_1(d)$" for the AS-serializability violation from the program shown in Listing 2. For a generic problematic access pattern p, the definition of the NFA $\mathcal{A}_{\mathsf{mon}}$ that recognizes traces containing p follows naturally.

Fig. 3 gives a graphical depiction of $\mathcal{A}_{12}$. The initial state is q_1 and the final state is q_7. For a trace to be accepted by $\mathcal{A}_{12}$, it must make a transition through each state $q_{1\text{–}7}$. That is, the states $q_{1\text{–}7}$ track the memory accesses that make up problematic access pattern 12. The transition $(q_1, \mathsf{alloc}, q_2)$ models the allocation of the randomly-isolated object. Once the randomly-isolated object has been allocated, $\mathcal{A}_{12}$ ignores reads and writes by following the self-loop on state q_2 until the target EML process Proc begins a unit of work—modeled by the transition $(q_2, [, q_3)$.

$\mathcal{A}_{12}$ makes use of non-determinism. For each state q_i, $3 \leq i \leq 6$, there is a self-loop labeled with $[R_i W_i$. The symbol [models reentrant calls to units of work. The state does not change because Proc must be executing a unit of work for $\mathcal{A}_{12}$ to be in state q_i. The symbols R_i and W_i denote read and write accesses to any memory location by either thread. The use of non-determinism enables $\mathcal{A}_{12}$ to "guess" which memory accesses are actually involved in problematic access pattern 12. For example, if $\mathcal{A}_{12}$ is in state q_3 and observes the action $R_1(c)$, it can make a transition to state q_4—-the action is part of problematic access pattern 12—or it can follow the self-loop and remain in state q_3—the action is not part of problematic access pattern 12. Non-determinism is required because a thread may perform multiple memory accesses during a unit of work.

8.5 Queries

Once a multi-PDS Π and IPA $\mathcal{A}_{\mathsf{mon}}$ have been generated for an EML program EProg, a query is given to the model checker IPAMC to determine if an interleaved execution of Π is recognized by $\mathcal{A}_{\mathsf{mon}}$. If IPAMC returns true, meaning that there is an interleaved execution of Π recognized by $\mathcal{A}_{\mathsf{mon}}$, then the EML program has an AS-serializability violation. Otherwise, IPAMC returns false, and the EML program does not have an AS-serializability violation for the given scenario defined over a single or double-memory location for scenarios 1–5 and 6–14, respectively, and the target EML process Proc.

9 Experiments

EMPIRE is implemented using the WALA (IBM, 2009) program-analysis framework. Random isolation is implemented using WALA's facilities for rewriting the abstract-syntax tree (AST) of a Java program. The default object-sensitive call graph construction and points-to analyses are modified to implement the semantic reinterpretation of "`is_ri`", as described in §6.1. After rewriting the ASTs, EMPIRE emits an EML program from the input Java program. The EML program is then translated into multiple multi-PDS/IPA pairs, for which reachability queries are answered using IPAMC (Kidd et al., 2009b, 2011). All experiments were run on a dual-core 3 GHz Pentium Xeon processor with 4 GB of memory.

The goal of the experiments was to determine whether the techniques developed and implemented in EMPIRE could detect both single- and multi-location AS-serializability violations. We evaluated EMPIRE on eight programs from the ConTest suite (Eytani et al., 2007), which is a set of small benchmarks with known concurrency bugs. EMPIRE requires that the allocation site of interest be annotated in the source program. We annotated eleven of the twenty-seven programs that ConTest documentation identifies as having "non-atomic" bugs. Our front-end currently handles eight of the eleven (the AST rewriting currently does not support certain Java constructs). When analyzing a program with the user-specified allocation site SITE that allocates an object of type T, we used the default assumptions that (i) all fields declared by T are in one atomic set, and (ii) each public method defined by T is a unit of work.

To reduce the size of the generated models, we made minor modifications to the benchmark programs. For the programs analyzed, file I/O is used to output debugging and scheduling information, and to receive input that specifies the number of threads. We removed these operations, and manually unrolled loops that allocate `Thread` objects 2 times. When a benchmark used a shared object of type `java.lang.Object` as a lock, the type was changed to `java.lang.Integer` because our implementation uses *selective* object-sensitivity, for which the use of `java.lang.Object` as a shared lock removes all se-

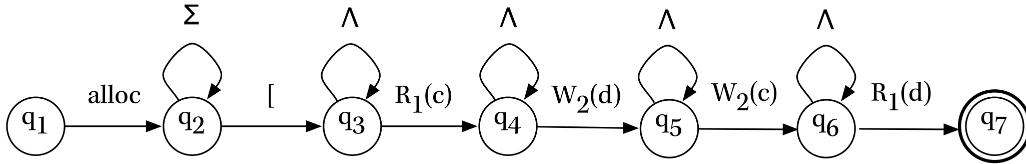

Fig. 3. The NFA $\mathcal{A}_{12}$ that recognizes traces of interleaved read and write memory accesses containing problematic access pattern 12 for the program shown in Listing 2. The edge labeled alloc denotes allocating the randomly-isolated object. An edge labeled $R_1(c)$ ($W_2(c)$) denotes a read from (write to) the field `Stack.count` by thread T_1 (T_2). Similarly, edges labeled $R_1(d)$ and $W_2(d)$ denote accesses to the field `Stack.data`. The symbols [and] denote Proc beginning and ending a unit of work, respectively. Σ denotes the input alphabet of $\mathcal{A}_{12}$, and Λ is defined as $\Sigma \setminus \{]\}$. Once $\mathcal{A}_{12}$ guesses that a violation will occur by making a transition to state q_3, it must observe a violation before the unit-of-work end symbol "]" appears in a trace. Otherwise, it will become stuck in a state q_{3-6}.

Benchmark	# Queries	# Violations Found	# Queries Verified
Account	642	38	604
AirlineTickets	900	18	882
PingPong	384	35	349
ProducerConsumer	512	72	440
SoftwareVerificationHW	15	6	9
BugTester	615	0	615
BuggyProgram	615	16	599
shop	900	27	873
Totals	4583	212	4371

Table 6. For each benchmark, column "# Queries" gives the number of queries that were generated (i.e., the number of IPAs generated for a benchmark's model Π). Columns "# Violations Found" and "# Queries Verified" give the breakdown of query satisfaction (i.e., an execution of Π is and is not, respectively, recognized by a generated IPA). The horizontal line after row 4 separates the benchmarks that did not contain any synchronization operations after abstraction from those that still did contain synchronization operations.

lectivity and severely degrades performance.[14] The programs `SoftwareVerificationHW` and `shop` define each thread's `run`() method to consist of a loop that repeatedly executes one unit of work. For these programs, the code in the body of the loop was extracted out into its own method so that the default unit-of-work assumptions would be correct. Each modification had no impact on the AS-serializability violations that could occur in a benchmark. The difficulty in automating these transformations is determining programmer intent. For example, removing debugging output is easy for an analyst who can use comments, variable names, and generated output to determine that certain output statements are not relevant to program correctness. The use of Java annotations to decorate such statements would greatly simplify that task. That is, given annotations such as @`debug` or @`lock` on program statements and fields, respectively, manual transformations would no longer be necessary.

In total, EMPIRE generated 4583 IPAMC queries. Each query was analyzed using a 300-second timeout, and completed successfully within that threshold. Tab. 6 presents a summary of the analysis results. The dividing line that separates the first four benchmarks from the latter four benchmarks pertains to lock usage. Each generated EML program consists of a single lock, namely, the lock for the randomly-isolated object.[15] However, for the first four benchmarks, the code does not contain synchronized methods, and hence does not acquire and release the lock associated with the randomly-isolated object. For the latter four benchmarks, the code contains synchronized methods, and the generated EML programs for these four contain lock and unlock operations. In total, EMPIRE:

1. found 212 (possible) AS-serializability violations (col. "# Violations Found"); and
2. determined that 4371 queries did not contain an AS-serializability violation (col. "# Queries Verified").

Tab. 7 presents a breakdown of the problematic-access-pattern numbers of the AS-serializability violations that were found for each benchmark program. For 7 of the 8 benchmarks listed in Tab. 7, EMPIRE found multiple actual AS-serializability violations (indicated by ⊗), where by actual AS-serializability violation we mean a true AS-serializability violation in the concrete concurrent Java program listed. The false positives (denoted by ⊙) reported for `PingPong` are due to an over-approximation of a thread's control flow—exceptional-control-flow paths are allowed in the model that cannot occur

[14] By selective object-sensitivity, we mean that a full object-sensitive call graph is not constructed because doing so exhausts memory resources. Instead, the call graph is only object sensitive with respect to a few key object types: the type T of the specified allocation site, the types of all of T's fields, and the type of all subclasses of threads.

[15] EMPIRE can generate multi-lock EML programs when the Java program makes use of global locks that are guaranteed to be unique, such as the lock that is associated with a `Class` object.

during a real execution of the program. Tab. 7 entries marked by "?" indicate problematic access patterns for which there is an AS-serializability violation in the EML program, but has not been verified in the original Java program.[16]

Overall, the experiments show that EMPIRE is able to detect both single- and multi-location AS-serializability violations. It is interesting to note that for benchmarks where a multi-location AS-serializability violation was found, a single-location AS-serializability violation also occurred. This can be explained by the fact that Java methods typically read and write to a field multiple times, which allows for both single- and multi-location AS-serializability violations to occur.

10 Related Work

Strong updates on an isolated non-summary object. The idea of identifying non-summary objects so that strong updates can be performed on them has a long history in shape-analysis algorithms. Originally, some non-summary objects could be identified as a fortuitous byproduct of the abstraction in use. For instance, with approaches based on k-limiting (Jones and Muchnick, 1981; Horwitz et al., 1989) non-summary nodes are maintained along selector-edge paths of length $\leq k$ from program variables. The shape abstraction of Chase et al. (1990) tracked which node merges in an abstract memory configuration preserved the property that the resulting abstract node represents a single concrete node in each store that the abstract memory configuration represents.

Plevyak et al. (1993) introduced the idea of deliberately *isolating* a non-summary node that represents only the memory location that will be updated during a transition. They called this operation *deconstruction*. Deconstruction has turned out to be essential, both for developing analyses that are tolerant of temporary violations of data-structure invariants, as well as for developing analyses that are capable of synthesizing data-structure invariants while analyzing loops that traverse linked data structures. Ordinarily, deconstruction is used to isolate the current node m being operated on, which is often the only node that does not satisfy some node-centric property $\varphi(n)$ that holds for all other nodes n_i of the data structure (often a local connectivity property, but sometimes a non-local reachability property). Once "surgery" has been completed on node m to establish $\varphi(m)$, the loop moves on to another node, and m is often folded into an existing summary node—and thus $\varphi(\cdot)$ continues to hold for *all* concrete nodes represented by the summary node. By this means, the fixed-point-finding loop of an abstract-interpretation solver can perform what amounts to an inductive argument that the loop (re-)establishes $\varphi(\cdot)$ throughout the data structure.

[16] Due to a limitation of our implementation in not producing witness traces for failed queries (sometimes called "counterexamples"), we are not able to check whether the AS-serializability violations found using IPAMC but not by our previous approach (Kidd et al., 2009a) using the model checker CPDSMC (Chaki et al., 2006) are actual bugs or false positives. The lack of witness traces is not a fundamental limitation of the approach used in IPAMC, just of our current implementation; in principle, it is possible to extend IPAMC to produce witness traces.

The work of Plevyak et al. (1993), and related work by Sagiv et al. (1998), exploited deconstruction in analyses tailored for *specific* data structures—doubly-linked lists in the case of Plevyak et al., and singly-linked lists in the case of Sagiv et al. Later work by Sagiv et al. (2002) gave an account of deconstruction in terms of logic, and showed that the essence of deconstruction involves a step, called *focus*, which forces the values of certain formulas (e.g., one that specifies which memory location will be updated during a transition) from an indefinite value (*unknown*) to a definite value (*true* or *false*). *Focus* is a semantic reduction (Cousot and Cousot, 1979): it has the effect of converting an abstract memory configuration into one or more abstract memory configurations that are more precise than the original one. Moreover, the approach taken by Sagiv et al. (2002) provided the ability to apply deconstruction in *every* abstraction specifiable using their *parametric* framework for shape analysis (which is implemented in the TVLA system (Lev-Ami and Sagiv, 2000)).

Gopan et al. (2004, 2005) also used the idea of working with a non-summary element so that strong updates could be performed, and showed how it could be applied to the analysis of programs that manipulate arrays.

Work in the type-theory community that uses similar ideas includes that of Walker and Morrisett (2000), Foster et al. (2002), Fahndrich and DeLine (2002), Aiken et al. (2003), Ahmed et al. (2007), and Rondon et al. (2010).

When the analysis methods discussed above also employ the allocation-site abstraction (Jones and Muchnick, 1982), each abstract memory configuration will have some bounded number of abstract nodes per allocation site.

Like random-isolation abstraction, *recency abstraction* (Balakrishnan and Reps, 2006) uses no more than *two* abstract blocks per allocation site SITE: a non-summary block MRAB[SITE], which represents the **m**ost-**r**ecently-**a**llocated **b**lock allocated at SITE, and a summary block NMRAB[SITE], which represents the **n**on-**m**ost-**r**ecently-**a**llocated **b**locks allocated at SITE. As the names indicate, recency abstraction is based on tracking a *temporal* property of a block b: the "is-the-most-recent-block-from-SITE(b)" property, which serves to isolate MRAB[b] from the blocks represented by NMRAB[b].

With *counter abstraction* (McMillan, 1999; Pnueli et al., 2002; Yavuz-Kahveci and Bultan, 2002), numeric information is attached to summary objects to characterize the number of concrete objects represented. The information on summary object u of abstract configuration S describes the number of concrete objects that are mapped to u in any concrete configuration that S represents. Counter abstraction has been used in the analysis of infinite-state systems (McMillan, 1999; Pnueli et al., 2002), as well as in shape analysis (Yavuz-Kahveci and Bultan, 2002).

In contrast to all of the aforementioned work, random-isolation abstraction is based on isolating a *random* individual (so that strong updates can be performed on its abstract counterpart), tracking its properties, and generalizing from the properties of the randomly chosen individual to *all* of the individuals allocated at the same allocation site, according to Random-Isolation Principle 2. The isolation can be performed

Program	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Account	⊗	⊗	?	⊗	⊗						?	?	?	
AirlineTickets	⊗	⊗		⊗		⊗	⊗	⊗			?	?	?	
PingPong	⊗	⊗	⊗		⊗	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙
ProducerConsumer	⊗	⊗	⊗	⊗	⊗	?	?	?	?	?	⊗	⊗	⊗	?
SoftwareVerificationHW	?	?	?		?									
BugTester														
BuggyProgram		⊗		⊗							⊗	⊗	⊗	
shop	⊗	⊗		?		?	?	?	?	?	⊗	?	?	?

Table 7. Marked entries denote violations reported by EMPIRE, with ⊗ being a true positive; ⊙ a false positive; and ? being a reported violation that we have not determined whether or not it is a true positive. Scenarios 6–16 involve two memory locations.

either via (i) a source-to-source transformation of the original program (as explained in §6.1); (ii) in the concrete operational semantics; or (iii) in the abstract semantics. Method (i) facilitates the use of random isolation in conjunction with existing analyses as a way to strengthen them. In addition to the work described in the present paper, random isolation has been used to model check information-flow properties of decentralized information flow control (DIFC) systems, which manipulate potentially unbounded sets of processes, principals, and communication channels (Harris et al., 2009).

The work most closely related to random isolation is the technique of *temporal case splitting* (Emmi et al., 2009), which was developed independently and contemporaneously with random isolation (Kidd et al., 2009a). Temporal case splitting also provides a way to reduce analysis problems that involve an unbounded number of entities and resources to a finite abstraction. In temporal case splitting, manually introduced “Skolem variables” serve to name a single, arbitrary individual among a given class of individuals. Emmi et al. (2009) used temporal case splitting as one of several techniques in a tool to verify programs that use reference counting to track resources and schedule their deallocation.

Detection of concurrency-related bugs. Traditional work on error detection for concurrent programs has focused on classical data races. Static approaches for detecting data races include type systems, where the programmer indicates proper synchronization via type annotations (e.g., Boyapati et al. (2002)), model checking (e.g., Qadeer and Wu (2004)), and static analysis (e.g., Naik and Aiken (2007)). Dynamic analyses for detecting data races include those based on the lockset algorithm (Savage et al., 1997), on the happens-before relation (Min and Choi, 1991), or on a combination of the two (O’Callahan and Choi, 2003). A data race is a heuristic indication that a concurrency bug may exist, and does not directly correspond to a notion of program correctness. In our approach, we treat atomic-set, unit-of-work, and unitfor annotations as a specification language that allow a programmer to state his intensions more precisely. Moreover, AS-serializability accounts for multi-location consistency errors that the aforementioned techniques would not be able to find.

High-level data races may take the form of *view inconsistency* (Artho et al., 2003), where memory is read inconsistently, as well as *stale-value errors* (Burrows and Leino, 2004; Artho et al., 2004), where a value read from a shared variable is used beyond the synchronization scope in which it was acquired. Our problematic interleaving scenarios capture these forms of high-level data races, as well as several others, in one framework.

Several notions of serializability and associated detection tools have been presented, including Flanagan and Freund (2004); Sasturkar et al. (2005); Lu et al. (2006); Wang and Stoller (2006a). These correctness criteria ignore relationships that may exist between shared memory locations, and treat all locations as forming one atomic set. Therefore, they may not accurately reflect the intentions of the programmer for correct behavior. Atomic-set-serializability takes such relationships into account and provides a finer-grained correctness criterion for concurrent systems. For a detailed discussion and comparison of different notions of serializability see Hammer et al. (2008).

AS-serializability was proposed by Vaziri et al. (2006). That work focused on inference of locks. A dynamic violation-detection tool was proposed in Hammer et al. (2008) to find errors in legacy code. Hammer et al. also analyzed programs from the CONTEST benchmark suite, and their dynamic tool found AS-serializability violations in the programs `Account`, `AirlineTcks`, `BugTester`, `PingPong`, and `SoftwareVerificationHW`. EMPIRE is a static counterpart of their tool, with the benefit that EMPIRE considers multiple executions of a program (symbolically), instead of just one execution like the dynamic tool of Hammer et al. Even though EMPIRE did not find the AS-serializability violation in `BugTester` because EMPIRE exhausted its available resources, it did find AS-serializability violations in `shop`, which the tool of Hammer et al. did not find. The remaining two benchmarks were not analyzed by their tool.

Acknowledgements The authors would like to thank the anonymous reviewers for their valuable feedback.

References

Ahmed, A., Fluet, M., and Morrisett, G. (2007). L3: A linear language with locations. *Fundam. Inf.*, 77(4):397–449.

Aiken, A., Foster, J., Kodumal, J., and Terauchi, T. (2003). Checking and inferring local non-aliasing. In *PLDI*, pages 129–140.

Artho, C., Biere, A., and Havelund, K. (2004). Using block-local atomicity to detect stale-value concurrency errors. In *Proc. 2nd Int. Symposium on Automated Technology for Verification and Analysis (ATVA 2004)*, volume 3299 of *LNCS*, pages 150–164, Taipei, Taiwan. Springer.

Artho, C., Havelund, K., and Biere, A. (2003). High-level data races. *Journal on Software Testing, Verification and Reliability (STVR)*, 13(4):207–227.

Balakrishnan, G. and Reps, T. (2006). Recency-abstraction for heap-allocated storage. In *SAS*, pages 221–239.

Ball, T. and Rajamani, S. (2001). Automatically validating temporal safety properties of interfaces. In *SPIN*, pages 103–122.

Boyapati, C., Lee, R., and Rinard, M. (2002). Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, pages 211–230.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. (1990). Symbolic model checking: 10^{20} states and beyond. In *LICS*, pages 428–239.

Burrows, M. and Leino, K. R. M. (2004). Finding stale-value errors in concurrent programs. *Conc. and Comp.: Prac. and Exp.*, 16(12):1161–1172.

Chaki, S., Clarke, E., Kidd, N., Reps, T., and Touili, T. (2006). Verifying concurrent message-passing C programs with recursive calls. In *TACAS*, pages 334–349.

Chase, D., Wegman, M., and Zadeck, F. (1990). Analysis of pointers and structures. In *PLDI*, pages 296–310.

Cousot, P. and Cousot, R. (1979). Systematic design of program analysis frameworks. In *POPL*, pages 269–282.

Dillig, I., Dillig, T., and Aiken, A. (2010). Fluid updates: Beyond strong vs. weak updates. In *ESOP*, pages 246–266.

Emmi, M., Jhala, R., Kohler, E., and Majumdar, R. (2009). Verifying reference counting implementations. In *TACAS*, pages 352–367.

Eytani, Y., Havelund, K., Stoller, S. D., and Ur, S. (2007). Towards a framework and a benchmark for testing tools for multi-threaded programs. *Conc. and Comp.: Prac. and Exp.*, 19(3):267–279.

Fahndrich, M. and DeLine, R. (2002). Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24.

Flanagan, C. and Freund, S. N. (2004). Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267.

Foster, J., Terauchi, T., and Aiken, A. (2002). Flow-sensitive type qualifiers. In *PLDI*, pages 1–12.

Gopan, D., DiMaio, F., Dor, N., Reps, T. W., and Sagiv, S. (2004). Numeric domains with summarized dimensions. In *TACAS*, pages 512–529.

Gopan, D., Reps, T. W., and Sagiv, S. (2005). A framework for numeric analysis of array operations. In *POPL*, pages 338–350.

Gulwani, S., McCloskey, B., and Tiwari, A. (2008). Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246.

Hammer, C., Dolby, J., Vaziri, M., and Tip, F. (2008). Dynamic detection of atomic-set serializability violations. In *ICSE*, pages 231–240.

Harris, W., Kidd, N., Chaki, S., Jha, S., and Reps, T. (2009). Verifying information flow control over unbounded processes. In *FM*, pages 773–789.

Horwitz, S., Pfeiffer, P., and Reps, T. (1989). Dependence analysis for pointer variables. In *PLDI*, pages 28–40.

IBM (2009). Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/wiki/index.php`.

Jones, N. and Muchnick, S. (1981). Flow analysis and optimization of Lisp-like structures. In *Program Flow Analysis: Theory and Applications*, pages 66–74. Prentice-Hall.

Jones, N. and Muchnick, S. (1982). A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL*.

Kahlon, V. and Gupta, A. (2007). On the analysis of interacting pushdown systems. In *POPL*, pages 303–314.

Kidd, N. (2009). *Static Verification of Data-Consistency Properties*. PhD thesis, Univ. of Wisconsin-Madison.

Kidd, N., Lal, A., and Reps, T. (2008). Language strength reduction. In *SAS*, pages 283–298.

Kidd, N., Lammich, P., Touili, T., and Reps, T. (2011). A decision procedure for detecting atomicity violations for communicating processes with locks. *Int. J. Softw. Tools Tech. Transfer*, 13(1):37–60.

Kidd, N., Reps, T., Dolby, J., and Vaziri, M. (2009a). Finding concurrency-related bugs using random isolation. In *VMCAI*, pages 198–213.

Kidd, N. A., Lammich, P., Touili, T., and Reps, T. (2009b). A decision procedure for detecting atomicity violations for communicating processes with locks. In *SPIN*, pages 125–142.

Lahiri, S. K. and Qadeer, S. (2006). Verifying properties of well-founded linked lists. pages 115–126.

Lammich, P. and Müller-Olm, M. (2008). Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In *SAS*, pages 205–220.

Lev-Ami, T. and Sagiv, M. (2000). TVLA: A system for implementing static analyses. In *SAS*, pages 280–301.

Lu, S., Park, S., Seo, E., and Zhou, Y. (2008). Learning from mistakes—a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339.

Lu, S., Tucek, J., Qin, F., and Zhou, Y. (2006). AVIO: Detecting atomicity violations via access-interleaving invariants. In *ASPLOS*, pages 37–48.

McMillan, K. (1999). Verification of infinite state systems by compositional model checking. In *CHARME*, pages 219–234.

Milanova, A., Rountev, A., and Ryder, B. G. (2005). Parameterized object sensitivity for points-to analysis for Java. *TOSEM*, 14(1):1–41.

Min, S. L. and Choi, J.-D. (1991). An efficient cache-based access anomaly detection scheme. In *ASPLOS*, pages 235–244.

Naik, M. and Aiken, A. (2007). Conditional must not aliasing for static race detection. In *POPL*, pages 327–338.

O'Callahan, R. and Choi, J.-D. (2003). Hybrid dynamic data race detection. In *PPoPP*, pages 167–178.

Plevyak, J., Karamcheti, V., and Chien, A. (1993). Analysis of dynamic structures for efficient parallel execution. In Banerjee, U., Gelernter, D., Nicolau, A., and Padua, D., editors, *Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lec. Notes in Comp. Sci.*, pages 37–57, Portland, OR. Springer-Verlag.

Pnueli, A., Xu, J., and Zuck, L. (2002). Liveness with (0, 1, ∞)-counter abstraction. In *CAV*, pages 107–122.

Qadeer, S. and Wu, D. (2004). KISS: Keep It Simple and Sequential. In *PLDI*, pages 14–24.

Ramalingam, G. (2000). Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22:416–430.

Rondon, P., Kawaguchi, M., and Jhala, R. (2010). Low-level liquid types. In *POPL*, pages 131–144.

Sagiv, M., Reps, T., and Wilhelm, R. (1998). Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50.

Sagiv, M., Reps, T., and Wilhelm, R. (2002). Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298.

Sasturkar, A., Agarwal, R., Wang, L., and Stoller, S. D. (2005). Automated type-based analysis of data races and atomicity. In *PPoPP*, pages 83–94.

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. E. (1997). Eraser: A dynamic data race detector for multithreaded programs. *Theoretical Computer Science*, 15(4):391–411.

Schwoon, S. (2002). *Model-Checking Pushdown Systems*. PhD thesis, TUM.

Vaziri, M., Tip, F., and Dolby, J. (2006). Associating synchronization constraints with data in an object-oriented language. In *POPL*, pages 334–345.

Walker, D. and Morrisett, J. (2000). Alias types for recursive data structures. In *Types in Compilation*, pages 177–206.

Wang, L. and Stoller, S. D. (2006a). Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP*, pages 137–146.

Wang, L. and Stoller, S. D. (2006b). Runtime analysis for atomicity of multithreaded programs. *IEEE Trans. on Soft. Engr.*, 32:93–110.

Yavuz-Kahveci, T. and Bultan, T. (2002). Automated verification of concurrent linked lists with counters. In *SAS*, pages 69–84.