Finding Concurrency-Related Bugs using Random Isolation

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

[1] University of Wisconsin {kidd,reps}@cs.wisc.edu
[2] IBM T.J. Watson Research Center {dolby,mvaziri}@us.ibm.com
[3] GrammaTech, Inc.

Abstract. This paper describes the methods used in EMPIRE, a tool to detect concurrency-related bugs, namely atomic-set serializability violations in Java programs. The correctness criterion is based on *atomic sets* of memory locations, which share a consistency property, and *units of work*, which preserve consistency when executed sequentially. EMPIRE checks that, for each atomic set, its units of work are serializable. This notion subsumes data races (single-location atomic sets), and serializability (all locations in one atomic set).

To obtain a sound, finite model of locking behavior for use in EMPIRE, we devised a new abstraction principle, *random isolation*, which allows *strong updates* to be performed on the abstract counterpart of each randomly-isolated object. This permits EMPIRE to track the status of a Java lock, even for programs that use an unbounded number of locks. The advantage of random isolation is that properties proved about a randomly-isolated object can be generalized to all objects allocated at the same site. We ran EMPIRE on eight programs from the ConTest benchmark suite, for which EMPIRE detected numerous violations.

1 Introduction

This paper describes the methods used in EMPIRE, a tool to detect atomic-set serializability violations in concurrent Java programs. *Atomic-set serializability* [1] is a data-centric correctness criterion for concurrent programs. It is based on the notion of an *atomic set* of memory locations, which specifies the *existence* of an invariant or consistency property. Associated with atomic sets are *units of work*, which preserve atomic-set consistency when executed sequentially. Atomic-set serializability means that, for each atomic set, its units of work are *serializable*, where an execution is serializable if it is equivalent to a serial execution in which each thread's units of work are executed with no interleavings from other threads.

Atomic-set serializability subsumes other correctness criteria for concurrent systems, such as data-race freedom (single-field atomic sets), and serializability (all fields in one atomic set). Such other criteria ignore the intended relationships that may exist between shared memory locations, and thus may not accurately reflect the intentions of the programmer about correct behavior.

Empire is a tool to statically detect atomic-set serializability violations (henceforth referred to as "violations") in concurrent Java programs. A key challenge that we faced was how to create a sound, finite model of a Java program's locking behavior that is capable of tracking the status of a Java lock, for programs that use an unbounded number of locks. To address this issue, we devised a new abstraction principle, *random isolation*, which has two key advantages:

1. It allows *strong updates* to be performed on the abstract counterparts of each randomly-isolated object, which permits Empire to track the status of the Java lock associated with a randomly-isolated object.
2. It allows properties proved about a randomly-isolated object to be generalized to *all* objects allocated at the same site.

Empire is based on the result that executions that are not atomic-set serializable can be characterized by a set of problematic interleaving scenarios [1]: an execution that is free of all of these scenarios is guaranteed to be atomic-set serializable.[4] In Empire, a problematic interleaving scenario with respect to a set of shared memory locations is used as an input specification to a model checker. Specifically, Empire translates a concurrent Java program into a communicating pushdown system (CPDS) [2, 3], and translates the scenario into a *violation monitor* that checks for the occurrence of the scenario, and runs concurrently with the other CPDS processes. Once the translation is performed, the generated CPDS is fed into a CPDS model checker [3].

Previous work [1] addressed the inference of synchronization and appropriate placement of locks, given annotations for atomic sets and units of work. A second paper [4] focused on legacy code and checking whether an existing multi-threaded program is appropriately synchronized, by dynamically detecting the occurrence of problematic interleaving scenarios. The work on Empire complements these other approaches by providing a method to statically check Java programs for problematic interleaving scenarios. Empire's checking algorithm uses the CPDS model checker's semi-decision procedure to (symbolically) consider multiple executions of the program. This is in contrast with the dynamic-detection approach [4], which only looks at one execution at a time.

Empire has two modes of operation. For code that satisfies certain properties,[4] it can verify the absence of violations. If the properties are not met, then it can miss errors, and thus operates as a bug detector, rather than a verification tool. The contributions of our work can be summarized as follows:

– We introduce a new abstraction principle, *random isolation*, which allows *strong updates* to be performed on the abstract counterparts of each *randomly-isolated object*. With this approach, properties proved about a randomly-isolated object can be generalized to all objects allocated at the same site. Random-isolation is a generic abstraction that should be applicable in many

[4] This result relies on an assumption that programs do not always satisfy: a unit of work that writes to one location of an atomic set, writes to all locations in that atomic set.

other contexts, such as typestate verification [5] and other temporal-safety analyses for object-oriented programs.
- We present a static technique for detecting atomic-set serializability violations in concurrent Java programs. The method uses random isolation to obtain a sound, finite model of locking behavior that, in many circumstances, is able to track the status of a Java lock precisely, even for programs that use an unbounded number of locks.
- We implemented these techniques in EMPIRE, and ran EMPIRE on eight programs from the ConTest benchmark suite[6], for which EMPIRE detected numerous violations, including ones involving multiple locations.

2 Overview

Fig. 1 is a simple Java program inspired by one of the ConTest programs [6]. There are two classes, `Shop` and `Client`. The intention of the programmer is that the method `Client.buy()` executes atomically, so that when `getItem()` is called on the parameter `Shop s`, `s` is non-empty. However, this intention is not implemented correctly: method `buy()` is synchronized on `this` and not on `s`, hence multiple clients of the same shop could interleave. Fig. 1 shows an interleaved program execution illustrating this concurency-related bug. After thread 2 finishes the call to `getItem()`, the field `items` is -1, which leads thread 1 to access the array `storage` with a negative index. This problem can be fixed by taking a lock on `s` in the body of `buy()`. Notice that there is no data race in this program, so traditional race detectors would not catch this bug.

This concurrency-related bug is an instance of an atomic-set-serializability violation. In this code, fields `items` and `storage` form an *atomic set*: they are meant to be updated atomically due to a consistency property. Each method of class `Shop` is a *unit of work* for this atomic set: when executed sequentially, it preserves the consistency property. In addition, the `buy()` method of `Client` must manipulate the parameter `Shop s` atomically. It is therefore a unit of work for the atomic set of `s`. The interleaved execution of Fig. 1 shows that the two units of work representing the method `buy()` are not serializable: i.e., the execution may produce a final state different from that of any serial execution of the two methods.

Atomic-set serializability is characterized by a set of problematic interleaving scenarios: i.e., an execution that does not contain any of the scenarios is atomic-set-serializable. In the example, the interleaved execution contains the following problematic scenario: $R_1(l_1), W_2(l_2), W_2(l_1), R_1(l_2)$, where l_1 (l_2) is bound to i (s). (See [1] for a complete list of these scenarios.) Notice that atomic-set-serializability is finer-grained than most notions of serializability because it is per atomic set, rather than embracing the whole heap.

EMPIRE detects atomic-set-serializability violations by statically checking for problematic interleaving scenarios. The user provides a concurrent Java program Prog, and specifies an allocation site ψ for a class T in Prog. (This is exemplified by the $\texttt{new}_\psi$ statement in Fig. 1 for the class `Shop`.) EMPIRE uses the default

assumptions of [4]: T has one atomic set containing all of T's declared fields (one atomic set per object), and every `public` method of T is a unit of work for that atomic set. Additionally, any method that takes a T object as a parameter is also a unit of work. EMPIRE then performs violation detection, focusing on the atomic sets of objects that can be allocated at ψ.

```
class Shop {
  Object[] storage = new Object[10];
  int items = -1;
  public static Shop makeShop(){
    return new_ψ Shop(); // ⟵ ψ
  }
  public synchronized Object getItem(){
    Object res = storage[items];
    storage[items--] = null;
    return res;
  }
  public synchronized void put(Object o){
    storage[++items] = o;
  }
  public synchronized boolean empty(){
    return (items == -1);
  }
}
```

```
class Client {
  public synchronized boolean buy(Shop s){
    if(!s.empty()) { s.getItem(); return true; }
    else return false;
  }
  public static Client makeClient(){
    return new Client();
  }
  public static void main(String[] args){
    Shop shop1 = Shop.makeShop();
    Shop shop2 = Shop.makeShop();
    Client client1 = makeClient();
    Client client2 = makeClient();
    new Thread("1") { client1.buy(shop1); }
    new Thread("2") { client2.buy(shop1); }
  }
}
```

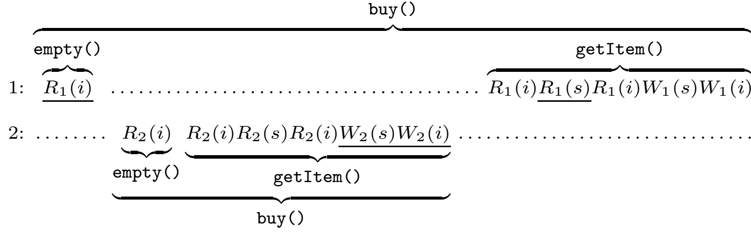

Fig. 1. Example Program. R and W denote a read and write access, respectively. i and s denote fields `item` and `storage`, respectively. Subscripts are thread ids.

EMPIRE performs violation detection in four stages. First, a source-to-source transformation is applied to the (potentially) infinite-data program Prog to prepare it for abstraction, obtaining a program Prog* (§3). Second, a finite-data abstraction is created for translating Prog* into EMPIRE's intermediate modeling language EML (§4). Third, from this EML program, EMPIRE generates CPDSs to model the program and monitor for problematic interleaving scenarios (§5). Fourth, state-space exploration is carried out on the generated CPDSs.

The challenge is to design a finite-data abstraction such that (i) the set of behaviors of the abstracted program is a sound overapproximation of the set of behaviors of the original Java program, and (ii) the abstraction is able to disallow certain thread interleavings by modeling the program's synchronization.

A natural choice for a finite-data abstraction is the *allocation-site abstraction* [7]. Given an allocation site ψ for class T, let $\mathsf{Conc}(\psi)$ denote the set of all concrete objects of type T that can be allocated at ψ. The allocation-site abstraction uses a single abstract object $\varsigma^\sharp_\psi$ to summarize all of the concrete

objects in $\mathsf{Conc}(\psi)$. Thus, for each field f defined by T, field $\varsigma^\sharp_\psi.f$ is a summary field for the set of fields $\{\varsigma.f \mid \varsigma \in \mathsf{Conc}(\psi)\}$. 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.

For such an approach 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. A *strong update* of the abstract state generally can only be performed when the analysis can prove that there is exactly one object allocated at ψ, i.e., $|\mathsf{Conc}(\psi)| = 1$.

Violation detection is concerned with tracking reads and writes to the fields of the T objects allocated at ψ. The allocation-site abstraction is a sound overapproximation for modeling reads and writes because a read (write) to the abstract field $\varsigma^\sharp_\psi.f$ corresponds to a possible read (write) to $\varsigma.f$, for all $\varsigma \in \mathsf{Conc}(\psi)$.

Violation detection must also model program synchronization. Empire accomplishes this by defining locks in the EML program that correspond to the objects of Prog^*. There are two possibilities for defining the semantics of an EML lock. The first 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 $\mathsf{Conc}(\psi)$, which in most circumstances would be unsound. In the example of Fig. 1, this interpretation of locking combined with the allocation-site abstraction would preclude the interleaved program execution that contains the bug, because the two `Client` objects would effectively get the same lock, and the two `buy()` methods would execute without interleaving. The second possibility for defining the semantics of EML locks is to interpret lock acquire as a *weak update*, i.e., the program may have acquired a particular lock. This semantics is sound, but the analysis gains no precision on locking behavior, since all lock operations are possible rather than definite. In general, this possibility would greatly increase the number of false positives. For instance, in the example of Fig. 1, if we were to fix the code by adding an additional synchronization block on `s` inside the body of `buy()`, analysis would still report a bug because locking behavior was modeled imprecisely.

Our solution is to use a new abstraction: *random-isolation abstraction*, which is a novel extension of allocation-site abstraction. The extension involves randomly isolating one of the concrete objects allocated at allocation site ψ and tracking it specially in the abstraction. Whereas allocation-site abstraction would associate one summary object to ψ, random isolation associates two objects to ψ: one summary and one non-summary. Because one is a non-summary object, it is safe to perform strong updates to its (abstract) state. The EML model will have an EML lock for each non-summary object, on which strong updates—definite lock acquires and releases—are performed. In constrast, because sound tracking of the lock state for a summary object generally would result in $\top$, our models have no locks on summary objects: their modeled behaviors are not restricted by synchronization primitives. This provides a sound, finite model of the locking behavior of Prog^*. (It is an over-approximation because the absence of locks on summary objects causes them to gain *additional* behaviors.)

The essence of random isolation can be captured via a simple source-to-source transformation. Consider the following code fragment.

$$\textsf{public static Shop makeShop() \{ return new}_{\psi}\ \textsf{Shop(); \}} \tag{1}$$

Random isolation involves transforming the allocation statement into

$$(\textsf{rand() \&\& test-and-set}(G_{\psi}))\ ?\ \textsf{new}_{\psi_{\star}}\ \textsf{Shop()} : \textsf{new}_{\psi}\ \textsf{Shop();}. \tag{2}$$

The site ψ from code fragment (1) is transformed into a conditional-allocation site, where the conditional "tests-and-sets" a newly introduced global flag G_{ψ}. The global flag G_{ψ} ensures that only one object can ever be allocated at the generated site $\psi_{\star}$. This has two benefits: (i) because abstract object $\varsigma^{\sharp}_{\psi_{\star}}$ is a non-summary object, strong updates can be performed on it, and (ii) because concrete object $\varsigma_{\psi_{\star}}$ is chosen randomly, every property proven to hold for $\varsigma^{\sharp}_{\psi_{\star}}$ must also hold for *every* concrete object $\varsigma_{\psi} \in \mathsf{Conc}(\psi)$.

3 Random-Isolation 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 ψ, $\mathsf{Conc}(\psi)$, cannot be distinguished by the allocation-site abstraction.*

Obs. 1 says that if one chooses to isolate a *random concrete* object ς from the summary object $\varsigma^{\sharp}_{\psi}$, the allocation-site abstraction would not be able to distinguish the randomly-chosen concrete object from any of the other concrete objects that are summarized by $\varsigma^{\sharp}_{\psi}$.

Random isolation extends allocation-site abstraction in two ways. First, whereas allocation-site abstraction uses one abstract object $\varsigma^{\sharp}_{\psi}$ to summarize the concrete objects $\mathsf{Conc}(\psi)$, random-isolation abstraction associates *two* abstract objects with ψ: $\varsigma^{\sharp}_{\psi}$ and $\varsigma^{\sharp}_{\psi_{\star}}$. Second, the global boolean flag G_{ψ} records whether the *randomly-isolated object* has been allocated or not. This eliminates the possibility that the concretization of the special abstract object $\varsigma^{\sharp}_{\psi_{\star}}$ is the empty set, and enforces isolation, which gives us *Random-Isolation Principle 1*:

Random-Isolation Principle 1 (Updates) *Let $\varsigma_{\star} \in \mathsf{Conc}(\psi)$ be a randomly-isolated concrete object. Because $\varsigma_{\star}$ is modeled by a special abstract object $\varsigma^{\sharp}_{\psi_{\star}}$, the random-isolation abstraction enables an analysis to perform strong updates on the state of $\varsigma^{\sharp}_{\psi_{\star}}$.*

Random isolation also provides a powerful methodology for proving properties of a program: a proof that a property ϕ holds for $\varsigma^{\sharp}_{\psi_{\star}}$ proves that ϕ holds for all $\varsigma \in \mathsf{Conc}(\psi)$. Consider a concrete trace of the program in which a concrete object ς' is allocated at a dynamic instance of ψ, and ϕ does not hold for ς'. Because of random isolation, the randomly-isolated object $\varsigma_{\psi_{\star}}$ is just as likely to be ς' as it is to be any other concrete object. Thus, the prover must consider

the case that $\varsigma_{\psi_\star}$ is ς'. Because the property holds for $\varsigma^\sharp_{\psi_\star}$, and because $\varsigma^\sharp_{\psi_\star}$ represents ς' in the trace under consideration, then the property must also hold for ς', which is a contradiction. This gives us *Random-Isolation Principle 2*:

Random-Isolation Principle 2 (Proofs) *Given a property ϕ and site ψ, a proof that ϕ holds for the randomly-isolated abstract object $\varsigma^\sharp_{\psi_\star}$ proves that ϕ holds for every object that is allocated at ψ. That is, $\phi(\varsigma^\sharp_{\psi_\star}) \rightarrow (\forall_{\varsigma \in \mathsf{Conc}(\psi)}.\phi(\varsigma))$.*

Before describing the technical details of how we implemented random isolation, we highlight the benefits of random isolation for performing violation detection. Because of random isolation, the state of the Java lock that is associated with the random instance $\varsigma_{\psi_\star}$ can be modeled precisely by the state of the special abstract object $\varsigma^\sharp_{\psi_\star}$. That is, the acquiring and releasing of the lock for $\varsigma_{\psi_\star}$ by a thread of execution can be modeled by a strong update on the state of $\varsigma^\sharp_{\psi_\star}$, thus allowing the analyzer to disallow certain thread interleavings when performing state-space exploration on the generated EML program.

3.1 Implementing Random Isolation

We implemented random isolation via the source-to-source transformation outlined in §2. To keep the source-to-source transformation semantics-preserving, *and* to ensure that only one concrete object can be allocated at $\psi_\star$, an atomic "test-and-set" operation must be performed on the boolean flag G_ψ.[5] Without the use of an atomic "test-and-set", the source-to-source transformation introduces a race condition that allows multiple objects to be allocated at $\psi_\star$. This in turn would invalidate *Random-Isolation Principles 1 & 2.*

While the use of a source-to-source transformation is not strictly necessary to implement random isolation, it allows existing object-sensitive analyses to be used with minimal changes. For example, let Pts be the points-to relation computed via a flow-insensitive, object-sensitive points-to analysis in the style of [8], and CG be an object-sensitive call graph.[6] Because these two analysis artifacts are object-sensitive, their respective dataflow facts make a distinction between those for $\psi_\star$ and those for ψ. For example, if T defines a method $T.m$, then CG will contain at least two nodes for $T.m$: one for object context $\psi_\star$, and one for object context ψ. Thus, inside of the control-flow graph for $T.m$ with object context $\psi_\star$, an analysis is able to take advantage of the fact that the special Java `this` variable is referring to the non-summary object $\varsigma^\sharp_{\psi_\star}$. That is, a unique context of $T.m$ has been created for $\varsigma^\sharp_{\psi_\star}$ without modifying the analyses!

[5] We use "test-and-set" to emphasize that random isolation is not particular to Java. For Java, we use the method `AtomicBoolean.compareAndSwap`.

[6] An object-sensitive call graph CG models the interprocedural control flow of a program: there is a node in CG for each method of the program for each context in which it can be invoked [8]. 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.

In some situations, however, a CG node's context is not enough to distinguish between $\varsigma^\sharp_{\psi_\star}$ and $\varsigma^\sharp_{\psi}$. Consider the code fragment "synchronized(t) { t.m() }", where t is defined as in code fragment (2), and $\mathsf{Pts(t)} = \{\varsigma^\sharp_{\psi_\star}, \varsigma^\sharp_{\psi}\}$. For performing violation detection, we require the ability to reason precisely about the state of a lock. Thus, in the program abstraction, we must be able to distinguish between the case when t references $\varsigma^\sharp_{\psi_\star}$ and when t references $\varsigma^\sharp_{\psi}$.

We solve this via a second source-to-source transformation that dispatches on the set of objects that are in Pts(t).

```
if (is_ri(t)) { synchronized(t) { t.m() } } else { synchronized(t) { t.m(); } }
```

In the source program, the method "is_ri" is defined as the identity function, and thus has no effect on the meaning of the program. However, the points-to analysis uses semantic reinterpretation of is_ri that performs a case analysis on Pts(t). Specifically, the reinterpreted is_ri performs the abstract test "$\mathsf{t} == \varsigma^\sharp_{\psi_\star}$", which allows the points-to analysis to perform *assume* statements on the branching paths (e.g., when following the true branch of the condition, the points-to analysis performs an "assume $\mathsf{Pts(t)} = \{\varsigma^\sharp_{\psi_\star}\}$"). One can view this as a way to achieve object-sensitivity at the level of a program block instead of just at the method level. Although we presented this second transformation in the context of violation detection, it is a generic approach that can be applied wherever an analysis needs to distinguish between $\varsigma^\sharp_{\psi_\star}$ and $\varsigma^\sharp_{\psi}$ to perform a strong update.

4 Translation to the Empire Modeling Language (EML)

We now describe how Empire defines an EML program.

4.1 Empire Modeling Language

An EML program $\mathcal{E}$ consists of (i) a finite number of shared-memory locations; (ii) a finite number of reentrant locks; and (iii) a finite number of concurrently executing processes.

An EML lock is reentrant, meaning that the lock can be reacquired by an EML process that currently owns the lock, and also that the lock must be released the same number of times 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 by Java's `synchronized` blocks and methods.

An EML process is defined by a set of (possibly) recursive functions, one of which is designated as the `main` function of the process. Each function consists of a sequence of statements, each of which is either a goto, choice, skip, call f, label *lab*, return, read m, write m, alloc l, lock l, unlock l, unitbegin, unitend, or

Java	EML	Condition
`x = o.f ; o.f = x`	read m_f ; write m_f	$\varsigma^{\sharp}_{\psi_\star} \in \mathsf{Pts}(o)$
`sync(o){...}`	lock $\varsigma^{\sharp}_{\psi_\star}$;...;unlock $\varsigma^{\sharp}_{\psi_\star}$	$\mathsf{Pts}(o) = \{\varsigma^{\sharp}_{\psi_\star}\}$
`sync(o){...}`	skip;...;skip	$\mathsf{Pts}(o) \neq \{\varsigma^{\sharp}_{\psi_\star}\}$
`o.start()`	start P_{ψ_θ}	$P_{\psi_\theta} \in \mathsf{Pts}(o)$, `Thread.start()` invoked.
`o.u()`	unitbegin;call u; unitend	$\varsigma^{\sharp}_{\psi_\star} \in \mathsf{Pts}(o)$, `u()` is a unit of work
`o.m()`	call m	

Table 1. Example Java statements, their corresponding EML statements, and the condition necessary to generate the EML statement.

start P. The statement "start P" starts the EML process named P. This is used to model the fact that when a Java program begins, only one thread is executing the `main` method, and all other threads cannot begin execution until they have been started by an already executing thread. (The other kinds of statements should be self-explanatory.)

4.2 EML Generation

Empire defines the EML program $\mathcal{E}$ as follows. To model the randomly-isolated abstract object $\varsigma^{\sharp}_{\psi_\star}$, $\mathcal{E}$ defines a shared memory location m_f for each field f of the class T, and also an EML lock $\varsigma^{\sharp}_{\psi_\star}$ to model the lock associated with $\varsigma^{\sharp}_{\psi_\star}$. The status of the global flag G_ψ is modeled by the EML lock $\varsigma^{\sharp}_{\psi_\star}$ being allocated or not. Let Threads be the set of all subclasses of `java.lang.Thread`. For each $\theta \in$ Threads, and for each allocation site ψ_θ that allocates an instance of θ, $\mathcal{E}$ defines an EML process P_{ψ_θ} that models the behavior of one instance of θ that is allocated at ψ_θ. Finally, $\mathcal{E}$ defines an EML process P_{main} that models the Java thread that begins execution of the `main` method. Each EML process P defines a function for each method that is reachable from P's entry point in CG. The translation from Java statements to EML statements is straightforward, with example translations given in Tab. 1.

5 Translation to Communicating Pushdown Systems

In this section, we describe the translation of EML programs into CPDSs.

5.1 Communicating Pushdown Systems

Definition 1. *A pushdown system (PDS) is a four-tuple* $\mathcal{P} = (Q, Act, \Gamma, \Delta)$*, where* Q *is a finite set of states,* Act *is a finite set of* actions*,* Γ *is a finite stack alphabet, and* Δ *is a finite set of rules of the form* $\langle q, \gamma\rangle \xrightarrow{a} \langle q', u'\rangle$*, where* $q, q' \in Q$*,* $a \in Act$*,* $\gamma \in \Gamma$*, and* $u' \in \Gamma^*$*. A configuration of* $\mathcal{P}$ *is a pair* $c = \langle q, u\rangle$*, where* $q \in Q$ *and* $u \in \Gamma^*$ *is the stack contents. A set of configurations* C *is regular if for each* $q \in Q$ *the language* $\{u \in \Gamma^* \mid \langle q, u\rangle \in C\}$ *is regular.*

Rule	Control flow modeled
$\langle q, n_1\rangle \overset{a}{\hookrightarrow} \langle q, n_2\rangle$	Intraprocedural edge $n_1 \rightarrow n_2$
$\langle q, c\rangle \overset{a}{\hookrightarrow} \langle q, e_f\ r\rangle$	Call to f from c that returns to r
$\langle q, x_f\rangle \overset{a}{\hookrightarrow} \langle q, \varepsilon\rangle$	Return from f at exit node x_f

Table 2. 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.

We assume that associated with each PDS $\mathcal{P}$ is an initial configuration c_{init}. For all $u \in \Gamma^*$, a configuration $c = \langle q, \gamma u\rangle$ can make a transition to a configuration $c' = \langle q', u'\,u\rangle$ if there exists a rule $r \in \Delta$ of the form $\langle q, \gamma\rangle \overset{a}{\hookrightarrow} \langle q', u'\rangle$. We denote this transition by $\xrightarrow{a}$ and extend it to $\xrightarrow{a_1 \cdots a_n}$ in the obvious manner. For a set of configurations C, we define the target language of $\mathcal{P}$ with respect to C as $\mathsf{Lang}(\mathcal{P}, C) = \{w \mid \exists c \in C, w \in \mathit{Act}^*, c_{\mathsf{init}} \xrightarrow{w} c\}$.

Because PDSs maintain a stack, they naturally model the interprocedural control flow of a thread of execution. The translation from a call graph and set of control-flow graphs (CFGs) into a PDS is shown in Tab. 2.

Definition 2. *A communicating pushdown system (CPDS) is a tuple* $\mathsf{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ *of PDSs. The action set Act of CP is equal to the union of the action sets of the* $\mathcal{P}_i$*, along with the special action* τ*:* τ *has the property that for all* $a \in \bigcup_1^n \mathit{Act}_i$*,* $\tau a = a\tau = a$*. The rules* Δ_i *for PDS* $\mathcal{P}_i$ *are augmented to include* $\{\langle q, \gamma\rangle \overset{a}{\hookrightarrow} \langle q, \gamma\rangle \mid q \in Q_i, \gamma \in \Gamma_i, a \in (\mathit{Act} \setminus \mathit{Act}_i)\}$*.*

Given n sets of configurations $S = (C_1, \ldots, C_n)$, we define the target language of a CPDS CP with respect to S as $\mathsf{Lang}(\mathsf{CP}, S) = \bigcap_{1 \le i \le n} \mathsf{Lang}(\mathcal{P}_i, C_i)$, where intersection enforces that all the $\mathcal{P}_i$ synchronize on the global actions. The goal of the CPDS model checker [3] is to determine if $\mathsf{Lang}(\mathsf{CP}, S)$ is empty. Because each language $\mathsf{Lang}(\mathcal{P}_i, C_i)$ can be, in general, a context-free language, and the problem of checking their intersection for emptiness is known to be undecidable, the CPDS model checker algorithm is only a *semi-decision* procedure. The semi-decision procedure may not terminate, but is guaranteed to terminate if there exists a finite-length sequence of actions, $w = a_1 \ldots a_n$, such that $w \in \mathsf{Lang}(\mathsf{CP}, S)$. Additionally, in some cases, the semi-decision procedure can determine that $\mathsf{Lang}(\mathsf{CP}, S) = \emptyset$. We refer the reader to [3] for more details.

5.2 CPDS Generation

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}. Empire generates a number of CPDSs for a given EML program: a CPDS is generated for each pair $(m_f, m_g) \in S_{\mathsf{Mem}} \times S_{\mathsf{Mem}}$ 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 [4]. In total, Empire generates $O(14 * (|S_{\mathsf{Mem}}|^2))$ CPDSs for an EML program.

For a generated CPDS CP, there is a PDS for each global component of the EML program: CP contains a PDS that monitors for a violation, a PDS for each

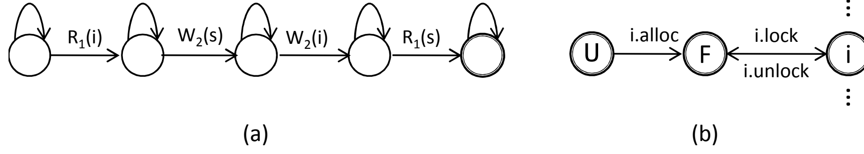

Fig. 2. (a) Race automaton for interleaving scenario 12 for example program in Fig. 1. (b) Lock automaton template. There is a state "i" for each EML process in S_{Procs}.

lock, and a PDS for each EML process. We now describe the generation technique for each component in turn. When the target language of a PDS is regular, we define it in terms of a finite-state machine (FSM). (An FSM is a single-state PDS with no push or pop rules; the initial configuration describes the initial state; and the final set of configurations describes the accepting state(s) of the FSM.)

The *violation monitor* detects when one of the interleaving scenarios occurs during a unit of work. The violation monitor is defined by a *race automaton* [4], which is a finite automaton that contains one state for each access defined by the scenario; transitions between states that reflect that an access has occurred; and self-transitions on states for accesses that do not make the scenario progress. Fig. 2(a) shows the race automaton that accepts the violation of scenario 12 for the example program.

Because an EML lock is reentrant, the language of the PDS that describes such behavior is context-free. However, previous work by the authors [9] developed a technique that safely removes reentrant acquisitions from an EML process, enabling the EML lock to be modeled as an FSM. Fig. 2(b) depicts a template FSM for one EML lock. The FSM begins in the Unallocated state, transitions to the Free state upon being allocated, and alternates between an "acquired-by-process-*i*" state and the Free state. Transitioning from Unallocated to Free denotes setting the global flag G_ψ associated with $\varsigma^\sharp_{\psi_\star}$.

Generating a PDS $\mathcal{P}$ for an EML process P is performed in two stages. First, a single-state PDS $\mathcal{P}_1 = (Q_1, Act_1, \Gamma_1, \Delta_1)$ is generated using the rule templates depicted in Tab. 2, with Act_1 being the set of all distinct EML statements used by P. $\mathcal{P}_1$ captures the interprocedural control flow of P.

Second, PDS $\mathcal{P}_2 = (Q_2, Act_2, \Gamma_2, \Delta_2)$ is defined as follows: $Q_2 = 2^{S_{\mathsf{Locks}}} \times Q_1$, $Act_2 = \{P.a \mid a \in (Act_1 \setminus \mathsf{start})\} \cup \{P'.\mathsf{alloc}\ \varsigma^\sharp_\star \mid P' \in (S_{\mathsf{Procs}} \setminus \{P\})\}$, $\Gamma_2 = \Gamma_1 \cup \{\mathsf{guess}\}$, and Δ_2 is defined from Δ_1 as shown in Tab. 3. Attaching the EML process's name P to the actions in Act_2 enables the violation monitor and locks to know which EML process performs an action. In Tab. 3, 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 before $\varsigma^\sharp_{\psi_\star}$ 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 state (s, q), $s \subseteq S_{\mathsf{Locks}}$, guess that EML process $P' \in (S_{\mathsf{Procs}} \setminus \{P\})$ allocates a lock $\varsigma^\sharp_\star \in (S_{\mathsf{Locks}} \setminus s)$, and

Action a	Rule $\langle q,\gamma\rangle \xhookrightarrow{a} \langle q',w\rangle$	
τ, start $\mathcal{P}'$	$\{\ \langle (s,q),\gamma\rangle \xhookrightarrow{a} \langle (s,q'),w\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}}\ \}$
alloc $\varsigma^\sharp_\star$	$\{\ \langle (s,q),\gamma\rangle \xhookrightarrow{P.a} \langle (s',p'),w\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge \varsigma^\sharp_\star \notin s \wedge s' = s \cup \{\varsigma^\sharp_\star\}\ \}$
lock/unlock $\varsigma^\sharp_\star$	$\{\ \langle (s,q),\gamma\rangle \xhookrightarrow{P.a} \langle (s,q'),w\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge \varsigma^\sharp_\star \in s\ \}$
read/write m_f	$\{\ \langle (s,q),\gamma\rangle \xhookrightarrow{P.a} \langle (s,q'),w\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge \varsigma^\sharp_{\psi_\star} \in s\ \}$
ubegin/uend	$\{\ \langle (s,q),\gamma\rangle \xhookrightarrow{P.a} \langle (s,q'),w\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge \varsigma^\sharp_{\psi_\star} \in s\ \}$
$\star$	$\{\ \langle (s,q),\gamma\rangle \xhookrightarrow{\tau} \langle (s,q),\mathsf{guess}\ \gamma\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}}\ \}$
$\star$	$\{\ \langle (s,q),\ \mathsf{guess}\rangle \xhookrightarrow{P'.\mathsf{alloc}\ \varsigma^\sharp_\star} \langle (s',p),\epsilon\rangle$	$\mid s \in 2^{S_{\mathsf{Locks}}} \wedge \varsigma^\sharp_\star \notin s \wedge s' = s \cup \{\varsigma^\sharp_\star\}$ $\wedge\ P' \in (S_{\mathsf{Procs}} \setminus \{P\})\ \}$

Table 3. Each row defines a set of PDS rules that are necessary for modeling the allocation of locks (see §5.2).

return back to the caller in the new state $(s \cup \{\varsigma^\sharp_\star\}, q)$. The guessing rule is then labeled with action $P'.\mathsf{alloc}\ \varsigma^\sharp_\star$.

Once CP has been generated, a language-emptiness query is passed to the CPDS model checker. This requires defining the target set of configurations for each PDS $\mathcal{P}_i$. For a PDS whose target language is regular, the target set of configurations is defined by the FSM. For a PDS that describes an EML process, the target set of configurations is any configuration (i.e., $\{\langle q,u\rangle \mid q \in Q, u \in \Gamma^*\}$). Let S be the configuration sets for the PDSs. The language-emptiness query as defined is such that $\mathsf{Lang}(\mathsf{CP}, S) = \emptyset$ is true *if-and-only-if* the EML program cannot generate a trace accepted by the violation monitor.

6 Experiments

Empire is implemented using the WALA [10] program-analysis framework. Random isolation uses WALA's support for rewriting the abstract-syntax tree of a Java program. The default object-senstive call graph construction and points-to analyses are modified to implement the semantic reinterpretation of "is_ri", as described in §3.1.

We evaluated Empire on eight programs from the ConTest suite [6], which is a set of small benchmarks with known non-trivial concurrency bugs. All experiments were run on a dual-core 3 GHz Pentium Xeon processor with 16 GB of memory. The analyzed programs are modified versions of those in the ConTest suite. To reduce the size of the generated models, we removed all use of file I/O from the programs. 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 selectivity and severely degrades performance. The programs `AllocationV` and `Shop` define a thread's `run()` method that consists of a loop that repeatedly executes one unit of work. For these programs, the code body of the loop was extracted out into its own method so that the default unit-of-work assumptions would be correct. Finally,

Nr	Program	CPDSs	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	account	352	$\checkmark$	$\checkmark$		$\checkmark$	$\checkmark$									
2	airlinesTckts	630	$\checkmark$	$\checkmark$		$\checkmark$		$\checkmark$	$\checkmark$	$\checkmark$						
3	AllocationV	15	$\checkmark$	$\checkmark$												
4	BuggyProgram	68				$\checkmark$										
5	BugTester	435														
6	PingPong	460	$\checkmark$	$\checkmark$	$\checkmark$		$\checkmark$	$\not\checkmark$	$\not\checkmark$	$\not\checkmark$	$\not\checkmark$	$\not\checkmark$	$\not\checkmark$		$\not\checkmark$	$\not\checkmark$
7	ProdConsumer	291	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$		$\checkmark$		$\checkmark$	$\checkmark$
8	Shop	542	$\checkmark$	$\checkmark$		$\checkmark$							$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
	Totals	2793														

Table 4. Marked entries denote violations reported by EMPIRE, with $\checkmark$ being a verified violation and $\not\checkmark$ a false positive. Scenarios 6-11 involve two memory locations.

many benchmarks allocate threads in a loop. We manually unrolled these loops to make the programs use a finite number of threads.

For 6 of the 8 benchmarks listed in Tab. 4, EMPIRE found multiple violations. The false positives reported for `PingPong` are due to an overapproximation of a thread's control flow—exceptional control paths are allowed in the model that cannot occur during a real execution of the program. The program `ProducerConsumer` has an atomic set with mutiple fields and uses no synchronization. While not interesting for violation detection, it validates that our approach is able to detect each of the problematic interleaving scenarios. Overall, the initial results are encouraging for applying EMPIRE to larger programs. Future work on EMPIRE includes a thread-escape analysis—determining the allocation sites that allocate shared objects—which would allow EMPIRE to analyze the escaping allocation sites using the default assumptions.

7 Related Work

Strong updates on an isolated non-summary object. The idea of isolating a distinguished non-summary node that represents the memory location that will be updated during a transition, so that a strong update can be performed on it, has a long history in shape-analysis algorithms [11–13]. When these methods also employ the allocation-site abstraction, each abstract memory configuration will have some bounded number of abstract nodes per allocation site.

Like random-isolation abstraction, *recency abstraction* [14] uses no more than *two* abstract blocks per allocation site ψ: a non-summary block MRAB[ψ], which represents the **m**ost-**r**ecently-**a**llocated **b**lock allocated at ψ, and a summary block NMRAB[ψ], which represents the **n**on-**m**ost-**r**ecently-**a**llocated **b**locks allocated at ψ. As the names indicate, recency abstraction is based on tracking a *temporal* property of a block b: the is-the-most-recent-block-from-$\psi(b)$ property.

With counter abstraction [15–17], 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 to analyze infinite-state systems [15, 16], as well as in shape analysis [17].

In contrast to all of the aforementioned work, random-isolation abstraction is based on tracking the properties of a *random* individual, and generalizing from the properties of the randomly chosen individual according to Random-Isolation Principle 2.

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 (see e.g., [18]), model checking (see e.g., [19]), and static analysis (see e.g., [20]). Dynamic analyses for detecting data races include those based on the lockset algorithm [21], on the happens-before relation [22], or on a combination of the two [23]. 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 consider atomic-set serializability as a correctness criterion, which captures the programmer's intentions for correct behavior directly.

High-level data races may take the form of *view inconsistency* [24], where memory is read inconsistently, as well as *stale-value errors* [25], 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 (or atomicity) and associated detection tools have been presented, including [26–29]. 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 [4].

Atomic-set serializality was proposed by Vaziri et al. [1]. That work focused on inference of locks. A dynamic violation-detection tool was proposed in [4] to find errors in legacy code. Our tool is a static counterpart with the benefit that it (symbolically) considers multiple executions of a program, instead of just one execution like the dynamic tool.

References

1. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL. (2006)
2. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL. (2003)
3. Chaki, S., Clarke, E.M., Kidd, N., Reps, T.W., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: TACAS. (2006)
4. Hammer, C., Dolby, J., Vaziri, M., Tip, F.: Dynamic detection of atomic-set-serializability violations. In: ICSE. (2008)

5. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. TSE (1986)
6. Eytani, Y., Havelund, K., Stoller, S.D., Ur, S.: Towards a framework and a benchmark for testing tools for multi-threaded programs. Conc. and Comp.: Prac. and Exp. **19**(3) (2007)
7. Jones, N., Muchnick, S.: A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In: POPL. (1982)
8. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. TOSEM **14**(1) (2005)
9. Kidd, N., Lal, A., Reps, T.: Language strength reduction. In: SAS. (2008)
10. Watson Libraries for Analysis (WALA), T.J.: http://wala.sourceforge.net/wiki/index.php
11. Horwitz, S., Pfeiffer, P., Reps, T.: Dependence analysis for pointer variables. In: PLDI. (1989)
12. Jones, N., Muchnick, S.: Flow analysis and optimization of Lisp-like structures. In: Program Flow Analysis: Theory and Applications. Prentice-Hall (1981)
13. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems **24**(3) (2002)
14. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: SAS. (2006)
15. McMillan, K.: Verification of infinite state systems by compositional model checking. In: CHARME. (1999) 219–234
16. Pnueli, A., Xu, J., Zuck, L.: Liveness with $(0, 1, \infty)$-counter abstraction. In: CAV. (2002)
17. Yavuz-Kahveci, T., Bultan, T.: Automated verification of concurrent linked lists with counters. In: SAS. (2002) 69–84
18. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: OOPSLA. (2002)
19. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: PLDI. (2004)
20. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL. (2007)
21. von Praun, C., Gross, T.R.: Object race detection. In: OOPSLA. (2001)
22. Min, S.L., Choi, J.D.: An efficient cache-based access anomaly detection scheme. In: ASPLOS. (1991)
23. O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. In: PPoPP. (2003)
24. Artho, C., Havelund, K., Biere, A.: High-level data races. In: Proc. NDDL/VVEIS'03. (2003)
25. Burrows, M., Leino, K.R.M.: Finding stale-value errors in concurrent programs. Conc. and Comp.: Prac. and Exp. **16**(12) (2004)
26. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: POPL. (2004) 256–267
27. Sasturkar, A., Agarwal, R., Wang, L., Stoller, S.D.: Automated type-based analysis of data races and atomicity. In: PPoPP. (2005)
28. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: Detecting atomicity violations via access interleaving invariants. In: ASPLOS. (2006)
29. Wang, L., Stoller, S.D.: Accurate and efficient runtime detection of atomicity errors in concurrent programs. In: PPoPP. (2006)