EnforceMOP: A Runtime Property Enforcement System for Multithreaded Programs

Qingzhou Luo, Grigore Roşu
Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{qluo2, grosu}@illinois.edu

ABSTRACT

Multithreaded programs are hard to develop and test. In order for programs to avoid unexpected concurrent behaviors at runtime, for example data-races, synchronization mechanisms are typically used to enforce a safe subset of thread interleavings. Also, to test multithreaded programs, developers need to enforce the precise thread schedules that they want to test. These tasks are nontrivial and error prone.

This paper presents EnforceMOP, a framework for specifying and enforcing complex properties in multithreaded Java programs. A property is enforced at runtime by blocking the threads whose next actions would violate it. This way the remaining threads, whose execution is safe, can make global progress until the system eventually reaches a global state in which the blocked threads can be safely unblocked and allowed to execute. Users of EnforceMOP can specify the properties to be enforced using the expressive MOP multi-formalism notation, and can provide code to be executed at deadlock (when no thread is safe to continue).

EnforceMOP was used in two different kinds of applications. First, to enforce general properties in multithreaded programs, as a formal, semantic alternative to the existing rigid and sometimes expensive syntactic synchronization mechanisms. Second, to enforce testing desirable schedules in unit testing of multithreaded programs, as an alternative to the existing limited and often adhoc techniques. Results show that EnforceMOP is able to effectively express and enforce complex properties and schedules in both scenarios.

Keywords

Enforcement, Testing, Concurrency

1. INTRODUCTION

Multithreaded programs utilize multiple threads to accomplish jobs faster than sequential programs. However, multithreaded programs are afflicted with concurrency bugs. These bugs are caused by the inherent non-determinism in thread scheduling, so it is hard to detect and fix them. Although there is a large amount of work tackling this problem, ranging from static/dynamic analysis [10, 20, 21], to testing [15, 19, 27, 32, 38–40], and to state space exploration [23, 34], these have their own limitations: testing and analysis approaches suffer from false positives and negatives, and exploration does not scale and depends on program inputs.

On the other hand, runtime verification [9,24,41] combines formal methods and testing to check critical properties of a program dynamically. The key idea is that software system properties, often defined using temporal formalisms, can be used to generate program monitors. Any property violation is reported or resolved immediately rather than waiting for a bug to manifest. Runtime verification has been proven to be a promising technique to increase software reliability, with a large number of runtime verification techniques and tools developed, including Tracematch [1], PQL [33], PTQL [22], MOP [12] and Hawk/Eraser [16], among many others.

While runtime verification can effectively detect property violations, and sometimes even recover from such violations, unfortunately it provides no guarantee that properties are never violated. This is particularly problematic in multithreaded systems, where non-deterministic thread scheduling may hide potentially critical errors. For example, consider a concurrent database where one thread is in charge of authorizing users, and each user is assigned a thread for fetching data. The underlying property is that any user should be authorized before getting data, so for any given user the corresponding thread should wait until the first thread finishes authorizing. Runtime verification approaches can monitor the program execution and report violations of this property for each user, but *cannot* prove correctness: a successful run gives no guarantee that other runs, under different thread schedules, will also be successful.

The conventional approach is to employ language-specific synchronization mechanisms or adhoc sleep commands to enforce such properties when developing or testing multithreaded programs. For instance, Java provides a `synchronized` keyword, a `Thread.sleep()` method, and several other classes in the `java.util.concurrent` package. However, there are certain limitations when using these constructs to enforce arbitrary properties in multithreaded programs: (1) it is non-trivial and error-prone to use these constructs when the property to be enforced is complex, as shown later in this paper; and (2) all these constructs are mingled with the original program, so it is not modular and overall hard to identify and reason about the underlying properties that the developers are attempting to enforce.

In this paper we present EnforceMOP, a novel framework for enforcing complex properties in multithreaded programs. The properties are enforced at runtime and do not require to modify the source code, so they can be modularly maintained. We show that EnforceMOP can be used effectively both in developing and in testing multithreaded programs.

This paper makes the following specific contributions:

Technique: We propose a technique to enforce arbitrarily complex safety properties in multithreaded programs. The properties can be expressed using various formalisms.

Implementation: EnforceMOP is implemented in Java on top of JavaMOP [13], a state-of-the-art runtime verification framework. Following the philosophy of JavaMOP, EnforceMOP is implemented in a logic-independent way.

Evaluation: We evaluated the effectiveness of EnforceMOP in two aspects. First, as a framework to enforce general properties when developing multithreaded programs, specifically to enforce correct behavior of such programs. Second, as a testing framework to enforce thread schedules when unit testing multithreaded programs, specifically to enforce schedules in 185 existing multithreaded unit tests, and compared it with several existing testing frameworks.

Section 2 describes the usage of EnforceMOP on two real examples. Sections 3 describe the underlying techniques and implementation of EnforceMOP. Section 4 shows several applications of EnforceMOP and evaluates its effectiveness as a tool to specify schedules in unit testing multithreaded programs. We then discuss limitations and future work in Section 5, followed by related work and conclusion.

2. MOTIVATION

EnforceMOP can be used (1) to enforce general properties and (2) to enforce specific testing schedules in mutithreaded systems. Here we discuss two real world examples, one in each category, and show how EnforceMOP is used in each.

2.1 Enforcing General Properties

As stated in JavaDoc, an `ArrayList` in Java is not allowed to be iterated by an iterator and structurally modified at the same time [37].

> *The iterators returned by this class's iterator and listIterator methods are* fail-fast*: if the list is structurally modified at any time after the iterator is created, ...the iterator will throw a* `ConcurrentModificationException`*.*

However, it is very easy for developers to violate this. Moreover, it can be difficult to find and fix this error in multithreaded programs, because: (1) when using `ArrayList`, programmers are unaware of how it will be used in other threads; (2) the non-deterministic behavior of multithreaded programs makes it harder to reproduce and debug the problem. For example, as shown in a bug report in JFreeChart [35], one thread is iterating an `ArrayList` while another thread is attempting to call `add()` on the same `ArrayList` concurrently. As a result, an `ConcurrentModificationException` is thrown non-deterministically from the program.

We can easisly state the property of safe iteration in JavaMOP [12, 13], as shown in Figure 1 (ignore the gray areas for now, which are parts of the EnforceMOP extension). Monitoring-oriented programming (MOP) is a generic multi-formalism monitoring framework, which takes an implementation and a set of specifications as input, and checks whether the implementation violates the specifications at run time. JavaMOP is the Java instance of MOP, currently

```
1  enforce SafeList_Iteration(Collection c, Iterator i) {
2      creation event create after(Collection c) returning(Iterator i) :
3          call(Iterator Iterable+.iterator()) && target(c) {}
4
5      event modify before(Collection c) :
6          (
7            call(* Collection+.add*(..)) ||
8            call(* Collection+.clear(..)) ||
9            call(* Collection+.offer*(..)) ||
10           call(* Collection+.pop(..)) ||
11           call(* Collection+.push(..)) ||
12           call(* Collection+.remove*(..)) ||
13           call(* Collection+.retain*(..))
14         ) && target(c) {}
15
16     event next before(Iterator i) :
17         call(* Iterator.next(..)) && target(i) {}
18
19     event hasnextfalse after(Iterator i) returning(boolean b) :
20         call(* Iterator+.hasNext()) && target(i) && condition(!b) {}
21
22     fsm :
23         na [
24             create -> init
25         ]
26         init [
27             next -> unsafe
28             hasnextfalse -> safe
29         ]
30         unsafe [
31             next -> unsafe
32             hasnextfalse -> safe
33         ]
34         safe [
35             modify -> safe
36             hasnextfalse -> safe
37             next -> safe
38         ]
39
40         @nonfail {}
41
42         @deadlock { System.out.println("Deadlock detected!"); }
43 }
```

Figure 1: Safe List Iteration Specification

using AspectJ [30] for event specification and instrumentation. As shown in Figure 1, a JavaMOP specification consists of four parts. The first is the specification header, with modifiers and parameters. Each parameters instance yields a monitor instance. Here, the `Collection` and `Iterator` parameters indicate that a different monitor will be generated for each combination of instances of these two parameters. Monitors corresponding to different parameter instances will not interfere with each other. More details can be found in [12, 13]. The second part describes all the relevant events, which serve as an abstraction of the running program. Those events drive the monitor from one state to another state.

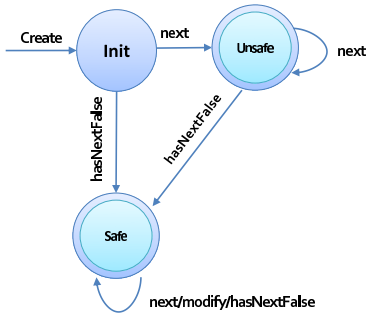

Figure 2: Safe List Iteration FSM

The third part is the actual property, starting with the *logic plugin* in which it is stated. Here we are using the finite state machine (FSM) plugin. Figure 2 depicts our property. A monitor begins with the `Init` state after an iterator is created for a specific `ArrayList` instance. Now if `next()` is called on the iterator then the monitor enters the `Unsafe` state. Any transitions not defined in the FSM will cause the monitor to enter a default `fail` state, indicating `ArrayList` was modified while an iteration is in progress. Method `hasNext()` returns false when the iterator has finished its job (we assume `hasNext()` is always called before `next()`, which is common practice), generating event `hasNextFalse` that makes the monitor enter its `Safe` state, indicating that modifications to the `ArrayList` are now allowed.

EnforceMOP has been purposely designed to require minimal learning effort from existing JavaMOP users. It should take less than one minute to change an existing JavaMOP specification into an EnforceMOP specification that enforces rather than monitors the former in multi-threaded systems. First, one needs to use the new `enforce` modifier (grayed in Figure 1). Second, one has to specify the desired state or group of states which the monitor should *not* be allowed to leave. Third, one may optionally use the new `@deadlock` handler to provide code to be executed in case of deadlock. We discuss the latter two in more detail below.

EnforceMOP enforces monitors to remain in certain states by controlling thread schedules. JavaMOP already allows users to associate code to monitor states, to be executed when the monitor reaches those states. Using the same notation, EnforceMOP enforces the monitor to never leave the specified states. Each logic plugin provides and documents its own monitor state names. The FSM plugin allows to define and name groups of states, and provides a predefined group of states named `nonfail` including all the states except `fail`. In our example, we state that we want EnforceMOP to never allow the monitors to leave their `nonfail` group of states. If a monitor attempts to execute a transition not shown in Figure 2, for example execute event `modify` in state `unsafe`, the thread scheduling code generated by EnforceMOP will block the unsafe thread and thus guarantee safe iteration behaviors. For example, when one thread is iterating the list so the monitor is in the `unsafe` state, any other thread attempting to modify the same list will get blocked until the end of the iteration is reached; then they are unblocked and allowed to perform their modifications.

Since threads in the program may get blocked by EnforceMOP, it is possible to cause deadlock in program directly or indirectly. For example, when the specified property is impossible to be enforced in a certain program (all possible thread schedules violate that property), all the threads will then be blocked by EnforceMOP thus resulting in a deadlock. In these cases, the `@deadlock` handler tells the monitor what to do when a deadlock occurs. Here we preferred to output an error message when a deadlock happens, but in general one can execute any code. For instance, shutdown the system, restart a certain thread, etc.

2.2 Enforcing Specific Testing Schedules

When writing a unit test for a multithreaded program, it is vital to have the ability to specify and enforce a desired thread schedule when running that test. Consider the real-life multithreaded test in Figure 3, borrowed from the TCK unit tests of `SynchronousQueue` in `java.util.concurrent`.

```
1  @Test
2  public void testPutWithTake() throws InterruptedException {
3      final SynchronousQueue q = new SynchronousQueue();
4      Thread t = new Thread(new CheckedRunnable() {
5              public void realRun() throws InterruptedException {
6                  int added = 0;
7                  try {
8                      while (true) {
9                          q.put(added);
10                         ++added;
11                     }
12                 } catch (InterruptedException success) {
13                     assertEquals("PutWithTake", 1, added);
14                 }
15             }}, "putThread");
16     t.start();
17     Thread.sleep(SHORT_DELAY_MS);
18     assertEquals("PutWithTake",0, q.take());
19     Thread.sleep(SHORT_DELAY_MS);
20     t.interrupt();
21     t.join();
22 }
```

Figure 3: Original SynchronousQueue Test in TCK

`SynchronousQueue` is a special kind of queue where the thread executing `put` blocks when the queue is full and the thread executing `take` blocks when the queue is empty. Thread `putThread` is calling `put` inside a loop to fill the queue. When the queue is full, `putThread` blocks. The desired thread schedule is: the main thread first waits for `putThread` to get blocked, then takes one element and checks it (line 18), then waits for `putThread` to get blocked again, and then interrupts it. This schedule is achieved in the TCK unit test using `sleep` statements, which as discussed in [27] and in Section 4.2 are non-modular, unreliable and slow.

EnforceMOP is an ideal vehicle to enforce specific testing schedules for multithreaded unit tests. The idea is to *separate* the functionality of the unit test from the desired schedule, and to implement the former as an unrestricted program (e.g., by removing the grayed sleep statements in Figure 3) and to enforce the latter with EnforceMOP. Figure 4 shows the EnforceMOP specification of the schedule meant in Figure 3. The event `beforeput` is generated right before calling method `put`, and events `beforeinterrupt` and `beforetake` right before calling methods `interrupt` and `take`, respectively. EnforceMOP defines a new pointcut, `threadBlocked`, telling the thread that is executing the event to *wait until* the specified thread is blocked. In this example, when the main thread is about to call the method `take` or `interrupt`, it waits until `putThread` gets blocked. We used the Extended Regular Expression (ERE) plugin (+ means one or more repetitions) to specify the actual schedule (line 19). Thus, the main thread blocks before it calls the method `take` until event `beforeput` occurs at least once and `putThread` blocks, then it unblocks and checks the assertion, and then it blocks again before it calls the `interrupt` until `beforeput` occurs and `putThread` blocks. The desired schedule is thus specified modularly, reliably and, as seen in Section 4.2, efficiently.

As discussed later in this paper, it is not easy to use existing multithreaded testing frameworks to specify this particular schedule, because it involves a loop. EnforceMOP is able to support repeating events in a thread schedule using the bare capabilities of the its logic plugins, e.g., the ERE +.

EnforceMOP has been implemented specification-formalism-independently and has been designed to support expressing and enforcing arbitrarily complex safety properties. The properties can be application-independent (such as the safe

```
1  enforce SynchronousQueueTest_testPutWithTake() {
2
3    String putThread = "";
4
5    event beforeinterrupt before() :
6      call(* Thread+.interrupt()) && threadBlocked(putThread){}
7
8    event beforetake before() :
9      call(* SynchronousQueue+.take()) && threadBlocked(putThread){}
10
11   event beforeput before() :
12     call(* SynchronousQueue+.put(..)) {
13             if (putThread.equals("")) {
14                 putThread = Thread.currentThread().getName();
15             }
16     }
17
18
19   ere : beforeput+ beforetake beforeput+ beforeinterrupt
20
21   @nonfail {}
22
23   @deadlock {System.out.println("Deadlock detected!");}
24 }
```

Figure 4: EnforceMOP Schedule for Test in Figure 3

list iteration property above) or application-specific (such as the specific schedule in the multithreaded unit test above). Different property specification formalisms have different expressiveness, and the flexibility to use any of them helps users specify a wide variety of properties precisely and elegantly. For example, as shown later, FSM cannot express some useful properties expressible with other formalisms. Additionally, EnforceMOP supports parametric specifications, so different (enforcing) monitor instances are created for different parameter instances.

EnforceMOP can be thought of as a *semantic*-based synchronization approach, complementary to the traditional *syntax*-based synchronization approach: the semantics is embodied in the formal specification for each property. EnforceMOP allows developers to declaratively and modularly state the actual properties they want to enforce in their programs, and thus by avoiding over-synchronization it has the potential to be more efficient than traditional synchronization mechanisms, as empirically shown later in this paper.

3. APPROACH AND IMPLEMENTATION

Here we give an overview of EnforceMOP, with particular emphasis on how it smoothly integrates with JavaMOP. The key challenge of this integration was to design the enforcement mechanisms in a formalism-independent way. Figure 5 recalls the overall architecture of JavaMOP. It consists of a Java-specific client and language-independent logic plugins. The logic plugin manager makes available to the client various logic plugins (discussed shortly), by taking as input a formula written in a specific logic and outputting language-independent monitoring pseudocode. This pseudocode is then used to generate Java and AspectJ code, which is finally waved with the original program to monitor.

3.1 Logic Plugins and Enforcement Categories

Each EnforceMOP specification requires a property over the specified events, formalized using one of the available logic plugins. Some formalisms are more convenient or more efficient than others in some situations. EnforceMOP currently supports for enforcement all the logic formalisms sup-

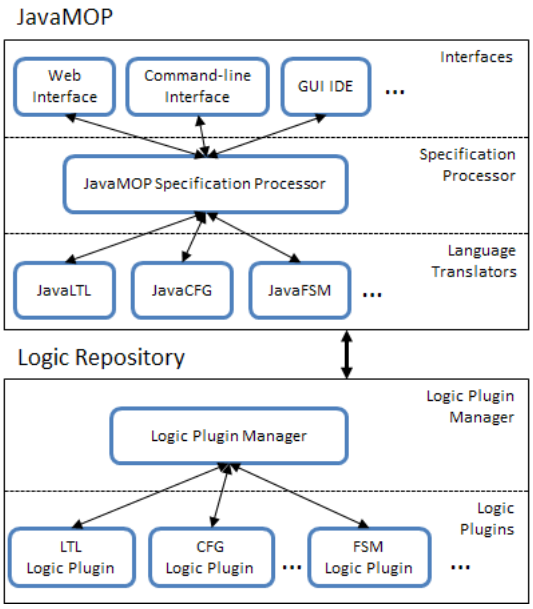

Figure 5: JavaMOP Overall Architecture

ported by JavaMOP for monitoring. We briefly recall them:

Finite State Machine (FSM): A finite state machine consisting of a set of states and a set of state transitions. Each transition is triggered by an event.

Extended Regular Expression (ERE): A regular expression extended with complement; each letter is an event.

Linear Temporal Logic (LTL): A future time linear temporal logic formula describing good or bad prefixes.

Past Time Linear Temporal Logic (PTLTL): A linear temporal logic formula with temporal operators referring to the past states of the execution trace.

Context Free Grammar (CFG): A context free grammar defined in BNF, where each terminal is an event.

String Rewriting System (SRS): Turing-complete string rewriting formalism, where each alphabet is an event.

Once a specific logic formalism is chosen, the next step is to choose in which way the property is enforced. For example, one can specify the correct behaviors of the system, and enforce the monitor to always obey the specification; alternatively, one can specify the adverse behaviors of a system, and enforce the monitor to never satisfy the specification. To accommodate all the existing logic plugins, EnforceMOP provides a set of pre-defined categories (a category can be viewed as a set of monitor states) to be enforced. As shown in Table 1, different logic formalisms have different corresponding categories. We describe each pre-defined category:

***fail*:** When the monitor encounters an event not accepted in current state (in FSM), or the current trace doesn't match any prefix of the given pattern (in ERE and CFG). In SRS, *fail* can be defined by users.

***nonfail*:** The opposite of *fail*, when the incoming event is accepted by the current state, or the current trace matches one prefix of the given pattern.

***succeed*:** In SRS, *succeed* is defined by users when certain patterns are matched.

Logic	Support Categories
FSM	*fail/nonfail*
ERE	*fail/nonfail/match/nonmatch*
LTL	*violation*
PTLTL	*violation/validation*
CFG	*fail/nonfail/match/nonmatch*
SRS	*fail/nonfail/succeed*

Table 1: Predefined Categories for each Logic Plugin

***match*:** Corresponds to a situation wherein the trace matches the entire specified pattern.

***nonmatch*:** Corresponds to a situation wherein the trace doesn't match the entire specified pattern.

***violation*:** Occurs when the trace is not a prefix of any trace that satisfies the given formula in LTL and PTLTL.

***validation*:** Corresponds to a situation wherein the trace satisfies the given formula in PTLTL.

Some plugins allow users to define their own categories, which can then be enforced using EnforceMOP. For example, FSM allows to define an alias of a group of states. EnforceMOP can then enforce the monitor to stay in one of those.

3.2 The Property Enforcing Algorithm

The key challenge in the design and development of EnforceMOP was to engineer its enforcement mechanism to work in a logic-formalism-independent way, to allow its users to choose any of the specification formalisms above for their properties and to enforce any of their categories. The problem is that different logic formalisms have different underlying representations of their monitors; for example, FSM uses lists of arrays to represent states and transitions, while CFG uses stacks to represent push down automata. However, all monitors share a common interface: take any given event and trigger a corresponding (logic-specific) transition.

The key idea of our monitor-independent enforcing algorithm is quite simple: use the common interface with a *clone* of the original monitor to decide whether to allow the current event to be executed on the original monitor, or to block the current thread. The algorithm is presented in Figure 6. The new event is sent to the cloned monitor, to check using its logic-specific semantics, which is irrelevant to EnforceMOP, whether the property we want to enforce would be violated if we let the event go through. If yes, then we block the current thread. If not, then it is safe for the original monitor to execute this event, so we let the event go through. We invoke the blocked thread and repeat the process above whenever a new event is generated in any other thread. Since a monitor is shared between different threads, its status may be changed by events executed in other threads. Whenever we find out that executing the pending event on the cloned monitor will not violate the property we want to enforce, we will unblock the thread and resume its execution.

3.3 Deadlock Detection

When enforcing a property, it could be possible that all the threads are blocked by EnforceMOP, so the program deadlocks. This happens when the program reaches a state in which any event to be executed by any thread would violate the property. Since property violations can mean anything depending upon the application and the property, our approach is to provide the mechanism and let the user decide how to use it, that is, how to proceed at deadlock. Specifically, EnforceMOP provides an on-the-fly deadlock detection mechanism which works as follows. Every newly started

```
1  // Inputs
2  Set<Category> violationCategories;
3  Event event;
4  Monitor origMonitor;
5
6  void enforceProperty() {
7      do {
8          clonedMonitor = origMonitor.clone();
9          clonedMonitor.execute(event);
10         if (clonedMonitor.status ∈ violationCategories) {
11             clonedMonitor = null; // for garbage collection
12             wait;
13         }
14         else {
15             clonedMonitor = null; // for garbage collection
16             break;
17         }
18     } while (true);
19     origMonitor.execute(event);
20     notify all waiting monitors;
21 }
```

Figure 6: Algorithm for Enforcing Properties

thread is recorded in a global map. A separate deadlock detection thread checks this map periodically. When all the threads in the map are blocked, a deadlock occurred. The *@deadlock* handler serves like any other JavaMOP handlers, so users can take arbitrary actions when a deadlock happens; for example, restart the system or print error messages.

3.4 Implementattion

We implemented EnforceMOP in Java as an extension of JavaMOP. JavaMOP takes a property file as input and generates an AspectJ file that contains monitor, recovery and instrumentation code, which is then compiled and waved within the original program using any AspectJ compiler. We added *enforce* as a new keyword modifier to JavaMOP properties, in a way that any existing JavaMOP property can be turned into an EnforceMOP by only adding the *enforce* modifier. To generate code to enforce a property, we extended the code generator in JavaMOP with a new class, `EnforceMonitor`, which is responsible for generating all the code to enforce a property when the *enforce* modifier is used.

As already noted in previous work on specifying thread schedules [27,40], it is crucial to have the ability to trigger an event when a specific thread gets blocked. For that reason, we added a new pointcut to EnforceMOP, `threadBlocked`. It takes a thread name as argument and triggers an event in the monitor only when that specific thread is blocked. We implemented this by using the `threadStart` pointcut of JavaMOP to add any thread to a global thread map when it starts. Then `threadBlocked` is easily implemented by polling the state of that specific thread in the map.

4. APPLICATIONS AND EVALUATION

We envision EnforceMOP to be used: (1) as a framework to enforce general complex safety properties at runtime; and (2) as a testing framework to enforce specific schedules when unit testing multithreaded applications. We next evaluate the effectiveness of EnforceMOP in these two aspects. We first present a number of applications using EnforceMOP to enforce general properties, then we use it to enforce specific testing thread schedules and compare it with several other multithreaded testing frameworks.

4.1 Enforcing General Properties

4.1.1 Safe Iteration

As shown in Figures 1 and 2, EnforceMOP can be used to guarantee safe iteration of a collection in multithreaded programs. Motivated by a real bug in JFreeChart [35], we used EnforceMOP to specify and enforce correct behaviors of iterating a collection in multithreaded programs. In the test case attached with the bug report, two threads are created and one of them adds a new element to the collection while the other iterates through the collection. These two actions are repeated many times, so in the original program the `ConcurrentModificationException` is thrown almost every time when the test case executes. After we applied the property in Figure 1 using EnforceMOP, the exception never gets thrown after 100 times of execution of the same test case.

4.1.2 Mutual Exclusivity

Another bug in JFreeChart [36] is caused by concurrent execution between any modification method and `hashCode` on the same `ArrayList`. The root cause of this bug is similar to the previous one: JDK's `hashCode` method iterates through all the elements of the list in order to compute the hash value of the whole list. So a `ConcurrentModificationException` will be thrown if `hashCode` and any other modification method are called at the same time. However, since the iteration of the list is encapsulated in `hashCode`, what users actually want is the mutual exclusivity *only* between the execution of `hashCode` and of other modification methods. This cannot be easily done using Java synchronization mechanisms. Suppose users only want the execution of `hashCode` and any other modification method to be mutually exclusive, but any other pairs of methods are allowed to execute concurrently. If we just use the `synchronized` keyword on all these then all the methods become mutually exclusive of each other, thus over-synchronizing the program and harming performance (for example, two threads could safely execute `hashCode` concurrently).

One can try to use a `ReadWriteLock` from `j.u.c` instead, for example to use `ReadLock` in `hashCode` and `WriteLock` in all modification methods. However, concurrent execution between any two modification methods would still be prohibited, thus reducing the potential for parallelism[1]. In fact, mutual exclusivity is a common property people want to enforce when writing multithreaded programs, but without careful consideration it is very easy to over-synchronize the entire program thus hurting the performance.

Figure 7 shows how to enforce mutual exclusivity for this specific case using EnforceMOP with the CFG plugin. The property is parametric in the list, so operations on different list instances will not interfere with each other. Since we want to enforce mutual exclusivity between method calls, we use both `before` and `after` pointcuts to describe events. There are four types of events in this property: `beforehashcode` and `afterhashcode` indicate the start and end of the execution of `hashCode`, and `beforemodify` and `aftermodify` represent the start and end of all the modification methods on `ArrayList`. The key part of the property is the logic formalism part. The property is defined using a CFG, which allows us to pair the start and the end events of the execution of `hashCode` or modification methods. While the execution of `hashCode` is in progress (event `afterhashcode` has not been encountered), the execution of any modification methods is not allowed (event `beforemodify` is not allowed).

[1]The concurrent use of `ArrayList` is known to be problematic; one should instead use concurrent data-structures from `j.u.c`. We use it here only to show how to enforce mutual exclusivity between groups of methods with EnforceMOP.

```
1  enforce SafeListCFG(List l) {
2
3      event beforehashcode before(List l) :
4          call(* Object+.hashCode(..)) && target(l) {}
5
6      event afterhashcode after(List l) :
7          call(* Object+.hashCode(..)) && target(l) {}
8
9      event beforemodify before(List l) :
10         (
11             call(* List+.add*(..)) ||
12             call(* List+.remove(..)) ||
13             call(* List+.retain*(..)) ||
14             call(* List+.clear(..)) ||
15             call(* List+.set*(..))
16         ) && target(l) {}
17
18     event aftermodify after(List l) :
19         (
20             call(* List+.add*(..)) ||
21             call(* List+.remove(..)) ||
22             call(* List+.retain*(..)) ||
23             call(* List+.clear(..)) ||
24             call(* List+.set*(..))
25         ) && target(l) {}
26
27     cfg :
28         S -> A S | B S | epsilon,
29         A -> A beforehashcode A afterhashcode | epsilon,
30         B -> B beforemodify B aftermodify | epsilon
31
32     @nonfail {}
33
34     @deadlock { System.out.println("Deadlock detected!"); }
35 }
```

Figure 7: Mutual Exclusivity between HashCode and List Modification Methods using CFG

Although the SRS plugin is the most expressive formalism available with EnforceMOP (it is Turing-complete), we often found it in our experiments that SRS is quite convenient to specify even simpler properties. For example, we can replace the CFG in Figure 7 with the following equivalent SRS:

```
srs :
    beforemodify aftermodify -> #epsilon .
    beforehashcode afterhashcode -> #epsilon .
    beforemodify afterhashcode -> #fail .
    beforehashcode aftermodify -> #fail .
    beforemodify beforehashcode -> #fail .
    beforehashcode beforemodify -> #fail .
```

The SRS rules apply on the trace as it is being generated to keep it in a canonical form. In our case, consecutive event pairs `beforehashcode` and `afterhashcode`, and `beforemodify` and `aftermodify`, will dissolve (`#epsilon` is the empty string), and the other four event pairs will force the monitor to fail. In Figure 7 we enforce the monitor to never enter its *fail* state (line 32), so whenever a thread wants to call a modification method while a `hashCode` method call is in progress, EnforceMOP will block that thread. Note that we only make `hashCode` and the group of the modification methods mutually exclusive, but no more than that. For example, the sequences `beforehashcode beforehashcode afterhashcode afterhashcode` and `beforemodify beforemodify aftermodify aftermodify` are both accepted. This allows maximum parallelism in the program. Note that this property cannot be expressed with FSM because the numbers of method start

```
1  enforce SafeAppendSRS(Category c) {
2
3      event beforeappend before(Category c) :
4          call(* Category+.append(..)) && target(c) {}
5
6      event afterappend after(Category c) :
7          call(* Category+.append(..)) && target(c) {}
8
9      event beforemodify before(Category c) :
10         (
11         call(* Category+.addAppender(..)) ||
12         call(* Category+.removeAppender(..)) ||
13         call(* Category+.removeAllAppenders(..))
14         ) && target(c) {}
15
16     event aftermodify after(Category c) :
17         (
18         call(* Category+.addAppender(..)) ||
19         call(* Category+.removeAppender(..)) ||
20         call(* Category+.removeAllAppenders(..))
21         ) && target(c) {}
22
23     srs :
24         beforemodify aftermodify -> #epsilon .
25         beforeappend afterappend -> #epsilon .
26         beforemodify afterappend -> #fail .
27         beforeappend aftermodify -> #fail .
28         beforemodify beforeappend -> #fail .
29         beforeappend beforemodify -> #fail .
30         beforemodify beforemodify -> #fail .
31
32     @nonfail {}
33
34     @deadlock { System.out.println("Deadlock detected!"); }
35 }
```

Figure 8: Mutual Exclusivity Property between Method Pairs in Log4J using SRS

and end events should match, and FSM does not have the expressiveness to count the number of occurrences of events. But it can be elegantly specified with CFG or SRS, showing the advantage of supporting multiple logic formalisms.

4.1.3 Read Write Lock

Here we address a performance problem in Log4J [7] caused by over-synchronization. The class `Category` is supposed to be thread safe, so the `synchronized` keyword is used in many of its methods (`append`, `addAppender` and `removeAppender`). Each `Category` object has a list of `appenders`; the method `append` calls methods on all the elements in the list but it does not modify the list itself. The `synchronized` keyword guarantees the mutual exclusivity between any methods, but it is not needed when two threads are both executing `append`. In the bug report one developer mentioned *"...observing plenty of threads waiting on this synchronization..."*.

We completely removed the usage of `synchronized` and used EnforceMOP instead to specify precisely the desired synchronization between those method pairs. The property is written using SRS and is shown in Figure 8. We first group the methods into two sets: methods that will not modify the list (`append`) and methods that will modify the list (`addAppender`, `removeAppender` and `removeAllAppenders`). Then we define events to mark the start and end of those methods. The property is similar to the previous one, except one more rule: `beforemodify beforemodify -> #fail`. This prevents two modification methods (e.g., `addAppender` and `removeAppender`) from happening in parallel, to avoid inconsistency. We disallow the parallel execution of any modification method and `append`, but we do allow parallel execution between methods `append`. This will increase the maximal parallelism and it is due to the intention of the developer.

No Sync	Original (Over-Sync)	EnforceMOP	`ReadWriteLock`
44.4	500.8	49.7	221.3

Table 2: Test execution time (ms) for different synchronization mechanisms

We wrote a test case to reproduce the performance problem caused by over-synchronization. We created 50 threads to run in parallel, half of them calling method `append` and another half calling methods `addAppender` and `removeAppender`. We collect running times with following configurations: the original over-synchronized code with `synchronized`; EnforceMOP enforcing proper synchronization as shown in Figure 8; a `ReadWriteLock` implementation proposed by the developer in the bug report; the original code with `synchronized` keyword removed completely, as a base line to show the performance overhead and it's incorrect. For each configuration we run the test case 10 times and get the average running time. Results are shown in Table 2.

From the results we can see EnforceMOP performs much better than the original over-synchronized version, since it increases the maximal parallelism of the application. Surprisingly, EnforceMOP also outperforms `ReadWriteLock`. The parallelism allowed by EnforceMOP is the same as with `ReadWriteLock`. We think the reason for our better performance is due to the fact that `ReadWriteLock` in Java involves calling a lot of library code and maintaining the lock status (since it's reentrant), while JavaMOP is highly optimized.

4.1.4 Dining Philosophers

Five philosophers sit next to each other around a round table. There are five forks placed between each pair of adjacent philosophers. Each philosopher needs to pick up the two forks around him to eat, and all of them are allowed to eat at the same time. Each fork can only be used by one philosopher at any time. A deadlock happens when each philosopher picks up a different fork at the same time, and all of them are attempting to pick the other to start eating.

To implement the dining philosophers problem, locks are typically used to enforce the property that one fork can only be taken by one philosopher at any time. Instead, we first use EnforceMOP to enforce this property, so no synchronization code is needed in the program at all. Then we enforce the property stating that at most four philosophers can eat at the same time, which guarantee deadlock freedom.

Synchronization Free Implementation

The sketch of our source is shown in Figure 9. Class `Phil` implements `Runnable`, so it can be run by a thread via the method `run`. Each philosopher grabs left fork first and right fork next, and then starts eating. Then he releases the left fork first and right fork second. Note that there's no synchronization or lock used in the source code. When all the philosophers are eating concurrently, it is possible that one fork is taken by multiple philosophers at the same time.

We use EnforceMOP to enforce the property of exclusive use of forks shown in Figure 10. This property is parametrized by a `Fork` instance. Event `occupy` corresponds to the start of method call `occupy` on a `Fork` instance and event `release` to the end of method call `release`. This property guarantees that when a fork is being used, it cannot be used again until released. Any other thread attempts to call `occupy` on a `Fork` instance at state `used` will be blocked. These allows us to implement the correct behavior of dining philosophers

```
1  public class Phil implements Runnable {
2
3      public Fork leftFork, rightFork;
4
5      public void getLeftFork() { leftFork.occupy(); }
6      public void releaseLeftFork() { leftFork.release(); }
7      ...
8
9      public void run() {
10         getLeftFork();
11         getRightFork();
12         eat();
13         releaseLeftFork();
14         releaseRightFork();
15     }
16 }
```

Figure 9: Source code of dining philosophers without synchronization

without any synchronization mechanism in the source code.

Figure 10: Exclusive use of Forks Property in FSM

Deadlock Free Property

The above property only guarantees the correct usage of forks. Deadlock is possible when each philosopher takes his left fork at the same time so no one is able to start eating. We use EnforceMOP to enforce a property to avoid deadlock in our implementation, as shown in Figure 11. The idea behind it is that we only allow four philosophers at most to eat at the same time, so at lease one philosopher would be able to grab both forks and finish eating. After that, he will release both forks and other philosophers will be able to eat. Event `useLeftFork` marks the start of method call `getLeftFork`, and event `releaseLeftFork` marks the end of method call `releaseRightFork`. Since in our implementation each philosopher will grab left fork first and release right fork last, these two events mark the beginning and end of actions of each philosopher. Each state serves as a *counter* of how many philosophers are eating now. At state `Four` any other philosophers attempt to grab forks will be blocked until a previous philosopher finishes eating and releases all his forks. With this property and the previous property in Figure 10, we are able to enforce the correct behavior of dining philosophers and avoid deadlocks at the same time, without using any synchronization in the source code. Compared with the previous property in Figure 10, this deadlock avoidance property is *not* parametric. It serves as a central coordinator to coordinate philosophers.

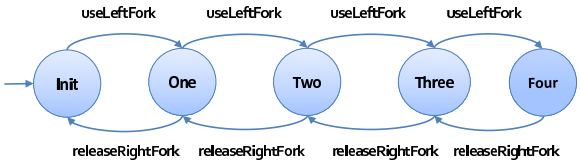

Figure 11: Deadlock Avoidance Property in FSM

4.1.5 *Fair Thread Scheduler*

In multithreaded programs *fairness* is a property of a thread scheduler which ensures every thread gets its turn

```
1  enforce FairScheduler(Task t) {
2
3      event workone before(Task t) :
4          call(* Task+.doWork(..)) && threadName("t1") && target(t) {}
5
6      event worktwo before(Task t) :
7          call(* Task+.doWork(..)) && threadName("t2") && target(t) {}
8
9      event afterwork after(Task t) :
10         call(* Task+.doWork(..)) && target(t) {}
11
12     fsm :
13         Init [
14                 workone -> ExecOne
15                 worktwo -> ExecTwo
16         ]
17         ExecOne [ afterwork -> OneDone ]
18         ExecTwo [ afterwork -> TwoDone ]
19         OneDone [ worktwo -> Finish ]
20         TwoDone [ workone -> Finish ]
21         Finish [ afterwork -> Init ]
22
23     @nonfail {}
24
25     @deadlock { System.out.println("Deadlock detected!"); }
26 }
```

Figure 12: Fair Scheduler Property

to execute eventually. In practice the lack of fairness may cause thread starvation bugs [8, 26].

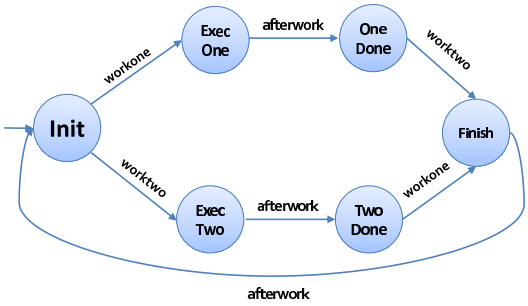

Figure 13: Fair Scheduler FSM

We use EnforceMOP here to enforce a simple fairness property in thread scheduling on an artificial example. In the underlying program there are two threads executing a loop concurrently. The numbers of loop iterations are same in both threads. In each loop iteration a method `doWork` will be called. Inside the `doWork` method each thread will sleep a random interval of time, and we use it to simulate real scenarios where workload is unknown. If we run the program without controlling the schedules, it is possible that one thread progresses much faster than another one. In an extreme situation, one thread may not even get scheduled to start running when another thread finishes. EnforceMOP is used to specify and enforce such a property that after one thread finishes one iteration of the loop (finish one time of execution of `doWork` method), it will until another thread finishes the same iteration and then starts its next iteration. As a result, the numbers of times of execution of `doWork` method in each thread won't differentiate each other more than one. We used FSM to specify this property, as shown in Figures 12 and 13. When one thread finishes executing one iteration of `doWork` (in state `OneDone` or `TwoDone`), it waits for another thread to finish one time of execution of `doWork`. After both threads finish, the monitor starts in `Init`

state again. After this property is enforced, each thread will always be in the same number of iterations in the loop.

Though it is an artificial example, we believe it shows the usefulness of EnforceMOP when specifying and enforcing complex properties. For instance, in a website where each user is served by a thread, it is important to guarantee no user would need to wait for a long time. With EnforceMOP it is possible to enforce such properties in a modular way without introducing any synchronization code in the system.

4.2 Enforcing Specific Testing Schedules

Multithreaded programs exhibit different behaviors under different thread schedules. Thus it is vital to have the ability to control thread schedules when performing unit testing. EnforceMOP can also be used as a testing framework to control thread schedules for each unit test. In that case, each property file is associated to some unit test, and serves as a thread schedule specification. In this section we first present our experience with using EnforceMOP as a testing framework to specify schedules in multithreaded unit tests. Then we compare EnforceMOP with several other testing frameworks for multithreaded programs.

4.2.1 Experience

To evaluate the effectiveness of using EnforceMOP as a testing framework, we took existing multithreaded unit tests and translated them to use EnforceMOP. Most of those tests used `sleep` or other synchronization mechanisms to control thread schedules. We first removed all the schedule control statements in those tests, and then wrote one property file for each testing schedule. We took the subject programs used in previous work [27, 40], and in total we translated 185 tests, as shown in Table 3.

When using EnforceMOP to specify thread schedules for a given unit test, the event sequences are already known and fixed. So in most cases there's no need to use complex logic formalisms; it is sufficient to only simply state the events to be executed in order. Indeed, we have used ERE in most of the cases, since the event sequences in a schedule is trivially an ERE expression. In some other cases, we have used PTLTL to specify schedules. PTLTL can be used to check whether a condition holds when a new event occurs, so it is suitable for enforcing the order between events.

Though most unit tests are quite simple, there are still cases where one event may occur many times. EnforceMOP is able to deal with repeating events. For example, making use of the * and + ERE constructs, properties can be expressed where an event can occur multiple times. More details on how EnforceMOP handle repeating events are mentioned in the comparison with IMUnit in Section 4.2.2.

4.2.2 Comparison with IMUnit

IMUnit [27] is a framework used to specify and control thread schedules in multithreaded unit tests. An event in IMUnit is fired explicitly by inserting a method call in the test code. A schedule in IMUnit is given as an annotation within a unit test, and it consists of several orderings between events. For example, `a -> b` specifies event `b` should only happen when event `a` has already happened. We compare EnforceMOP with IMUnit in the following aspects.

Expressiveness

IMUnit is also built upon JavaMOP, but its underlying schedule logic is a fragment of PTLTL which does not support repeating events. Consider the same example in Figure 3. With IMUnit we can insert events around the `put` method call, but since the method call is made inside a loop we cannot specify in the test schedule the exactly number of occurrences of an event. As already shown in Figure 4 with the help of operator + in ERE, EnforceMOP is able to express such schedules. The paper [27] mentioned that there were a few more tests where IMUnit was not able to express the schedules because of repeating events. We have successfully used EnforceMOP to specify and enforce the desired test schedules for all those cases. In fact, in our previous examples for specifying general properties, many events are repeating events and can happen anywhere in the program. Unlike IMUnit, EnforceMOP does not use the exact code location to specify an event; instead, it uses pointcuts to match for events. This way, EnforceMOP supports repeating events as long as the chosen logic plugin supports them.

Performance

The performance of IMUnit was evaluated by comparing the time to run all tests with the time to rune all the original sleep based tests. Since most of the sleep bases tests are over estimating the time needed for sleep operations, IMUnit tests were able to provide over 3x speed up over the original tests. We repeated the same set of experiments here. We used EnforceMOP to translate all the sleep based tests IMUnit used in experiments and calculated the speedup of using EnforceMOP to enforce schedules versus the original tests. The results are shown in table 4. We are able to achieve same or better speed up with EnforceMOP.

4.2.3 Comparison with MultithreadedTC

MultithreadedTC [40] is another unit testing framework, used to specify schedules in multithreaded tests using *ticks*. In multithreadedTC each test has to be written as a class, and each method in the class contains the code to be executed by a thread in a test. The test schedule is specified in terms of number of *ticks* with respect to a global logical clock. The method `waitForTick` takes a number as an argument, and it will block the current thread until the global clock reaches that number. The global clock will advance when all the threads are blocked.

Although MultithreadedTC is successfully applied on a number of tests [40], it requires users to change the original test a lot. EnforceMOP does not require users to change the original code at all, the schedule specification file (property) is in a separate file, so it is possible to have multiple schedules applied on a same test. Moreover, the schedule in MultithreadedTC is implicitly embedded using ticks. It may be non-trivial to infer a schedule from a MultithreadedTC test for a complicated test case. In terms of functionality, blocking events in MultithreadedTC are also implicit. Threads blocked by MultithreadedTC will be unblocked when all the threads are blocked. This makes it very hard to specify the scenario where one thread waits for another thread to get blocked using MultithreadedTC, while it is easy to do in EnforceMOP (and also in IMUnit).

4.2.4 Comparison with ConAn

ConAn [32] is a framework used to generate test driver code together with test schedules based on user provided scripts. Similar to MultithreadedTC, ConAn also employs *ticks* to specify logical time in thread schedules. A test in ConAn consists of a set of `#tick` blocks. Inside each `#tick`

Subject	Tests
Collections [2]	11
Hadoop [4]	1
JBoss-Cache [29]	20
Lucene [5]	2
Mina [6]	1
Pool [3]	2
Sysunit [14]	10
JSR-166 TCK [28]	138
$\sum$	185

Table 3: Subject Programs Statistics

Subject	Original	EnforceMOP	Speedup
Collections	2.22	0.26	8.54
Hadoop	1.39	0.38	3.66
JBoss-Cache	73.07	38.89	1.88
Lucene	10.78	2.87	3.76
Mina	0.24	0.14	1.71
Pool	1.48	0.076	19.47
Sysunit	14.47	0.30	48.23
JSR-166 TCK	16.67	6.48	2.57
GeometricMean			5.56

Table 4: Test execution time (s)

block there is a number of `#thread` blocks. Each `#thread` block contains the code that will be executed by a thread, and ConAn will generate tests based on that.

Since ConAn is also based on ticks, it also suffers from understandability when the schedule to be specified becomes complicated. Moreover ConAn does not support blocking events. Ticks in ConAn advance automatically after a fixed amount of time, making it unable to express certain schedules EnforceMOP and other frameworks are able to express.

5. DISCUSSION

EnforceMOP is implemented by cloning the monitor and executing the incoming event one step ahead using the clone. Depending upon the chosen logic formalism to express properties, it may be possible that "one step lookahead" is not enough and could cause a deadlock, even though the property is enforceable. For example, consider the ERE property `get* put`. When event `put` happens in one thread, EnforceMOP has no way to know whether event `get` will happen in the future or not (because code reachability is an undecidable problem). Executing event `put` as soon as it occurs will not violate the property, but if event `get` happens afterward then the monitor will deadlock because event sequence `put get` violates the property. In this case, the deadlock can be avoided if EnforceMOP blocks the thread executing event `put` until all the occurrences of event `get` have happened. To achieve this, it should be possible to add an *exploration* ability to EnforceMOP. Whenever a new event comes from a thread and both blocking the thread and not blocking it will not violate any property, record the current program location as a *choice point* and make a choice about whether to block that thread or not. After the current execution finishes, re-execute the program from the beginning until reaching the previous choice point, and make a different choice. In this way all the possible event sequences will be enumerated so it can be checked which event sequences will obey all the properties without causing deadlocks.

In our experience with using EnforceMOP so far, we have not seen many cases where exploration would be needed. Consequently, we leave it as future work to be investigated if we see more scenarios where exploration would be useful.

6. RELATED WORK

There is a rich body of work on runtime verification [1, 12, 13, 16, 25, 31, 33]. Most of them have hardwired specification languages. For example, Java-MaC [31] uses a customized language for interval temporal logic and PaX [25] only supports LTL. Moreover, all these runtime verification frameworks *monitor* rather than *enforce* properties. JavaMOP [12,13] is a parametric runtime verification framework which supports multiple logic formalisms. EnforceMOP is extending JavaMOP with the ability to enforce properties in multithreaded programs. Control theory techniques have been applied to steer program execution before a property is violated [18], but that is orthogonal to our approach in EnforceMOP; they did not aim to enforce properties in multithreaded programs by controlling thread schedules.

Many approaches have been proposed by researchers to test and verify multithreaded programs, such as static/-dynamic analysis [10, 20, 21], testing [15, 19, 27, 32, 38–40], and state space exploration [23, 34]. EnforceMOP does not aim to find bugs in multithreaded programs; rather, it is used as a testing framework to specify schedules in multithreaded unit tests. ConAn [32] uses a scripting language for specifying method sequences and test schedules to generate test driver code for multithreaded programs. MultithreadedTC [40] employed ticks to specify thread schedules. A domain-specific language to specify particular thread interleaving scenarios is proposed in [11], together with scenario-guided exploration of multithreaded programs. Our earlier work IMUnit [27] proposed a language together with event annotations to specify schedules in multithreaded unit tests. EnforceMOP supports all the features of the above frameworks, as described in Section 4.2. Moreover, with the underlying power of multiple logic formalisms, EnforceMOP can enforce complex schedules precisely and concisely.

A data-centric synchronization approach to avoiding certain concurrency errors such as data races is proposed in [17]. Their idea is to group fields into atomic sets and automatically enforce the atomicity when accessing those fields at runtime. EnforceMOP follows the same idea that synchronization in the program can be semantic rather than syntactic, but, with its different logic formalisms, EnforceMOP is able to express more complex properties than atomicity.

7. CONCLUSION

Multithreaded programs are hard to develop and test. In this paper we presented EnforceMOP, a novel framework for specifying and enforcing complex properties in multithreaded programs. We implemented EnforceMOP on top of JavaMOP, so it supports parametric properties and allows users to use various logic formalisms with different expressiveness. EnforceMOP is used in two kinds of applications. First, as a framework to enforce general properties which are not bound to any program locations or thread schedules. Second, as a testing framework to enforce specific thread schedules in unit tests, and we compared it with several other testing frameworks. Results showed that EnforceMOP is able to enforce both general properties and specific schedules effectively and efficiently.

8. REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor,

D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA*, 2005.

[2] Apache Software Foundation. Apache Commons Collections. `http://commons.apache.org/collections/`.

[3] Apache Software Foundation. Apache Commons Pool. `http://commons.apache.org/pool/`.

[4] Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org/`.

[5] Apache Software Foundation. Apache Lucene. `http://lucene.apache.org/`.

[6] Apache Software Foundation. Apache MINA. `http://mina.apache.org/`.

[7] Apache Software Foundation. LOG4J-50213. `https://issues.apache.org/bugzilla/show_bug.cgi?id=50213`.

[8] Apache Software Foundation. TOMCAT-25841. `https://issues.apache.org/bugzilla/show_bug.cgi?id=25841`.

[9] H. Barringer, B. Finkbeiner, Y. Gurevich, and H. Sipma, editors. *RV 05*, volume 144 of *ENTCS*, 2005.

[10] E. Bodden and K. Havelund. Racer: Effective race detection using AspectJ. In *ISSTA*, 2008.

[11] J. Burnim, T. Elmas, G. Necula, and K. Sen. CONCURRIT: Testing concurrent programs with programmable state-space exploration. In *HotPar*, 2012.

[12] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *TACAS*, 2005.

[13] F. Chen and G. Rosu. Mop: an efficient and generic runtime verification framework. In *OOPSLA*, 2007.

[14] Codehaus. Sysunit. `http://docs.codehaus.org/display/SYSUNIT/Home`.

[15] K. Coons, S. Burckhardt, and M. Musuvathi. Gambit: Effective unit testing for concurrency libraries. In *PPoPP*, 2010.

[16] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT SEN*, 30:1–7, 2005.

[17] J. Dolby, C. Hammer, D. Marino, F. Tip, M. Vaziri, and J. Vitek. A data-centric approach to synchronization. *ACM TOPLAS*, 34:4:1–4:48, 2012.

[18] A. Easwaran, S. Kannan, and O. Sokolsky. Steering of discrete event systems: Control theory approach. *ENTCS*, 144:21–39, 2006.

[19] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for Testing Multi-Threaded Java Programs. *CCPE*, 2003.

[20] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[21] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, 2008.

[22] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA*, 2005.

[23] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Journal on STTT*, 2:366–381, 2000.

[24] K. Havelund and G. Roşu, editors. *RV 01, RV 02, RV 04*, volume 55, 70, 113 of *ENTCS*, 2001, 2002, 2004.

[25] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *FMSD*, 2004.

[26] IBM. ECLIPSE-369251. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=369251`.

[27] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *FSE*, 2011.

[28] Java Community Process. JSR 166: Concurrency utilities. `http://g.oswego.edu/dl/concurrency-interest/`.

[29] JBoss Community. JBoss Cache. `http://www.jboss.org/jbosscache`.

[30] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[31] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance tool for Java programs. *ENTCS*, 55:218–235, 2001.

[32] B. Long, D. Hoffman, and P. Strooper. Tool Support for Testing Concurrent Java Components. *IEEE TSE*, 29:555–566, 2003.

[33] M. C. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, 2005.

[34] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[35] Object Refinery. JFREECHART-1051. `http://sourceforge.net/p/jfreechart/bugs/1051/`.

[36] Object Refinery. JFREECHART-187. `http://sourceforge.net/p/jfreechart/bugs/187/`.

[37] Oracle. JavaDoc ArrayList. `http://docs.oracle.com/javase/6/docs/api/java/util/ArrayLis`

[38] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[39] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. In *ICSE*, 2010.

[40] W. Pugh and N. Ayewah. Unit testing concurrent software. In *ASE*, 2007.

[41] O. Sokolsky and M. Viswanathan, editors. *RV 03*, volume 89 of *ENTCS*, 2003.