Run-time Assertion Checking of JML Annotations in Multithreaded Applications with e-OpenJML

Jorne Kandziora[1] Marieke Huisman[1*]
Christoph Bockisch[2] Marina Zaharieva-Stojanovski[1*]
[1]University of Twente, [2]Open University
the Netherlands

ABSTRACT

Run-time assertion checking of multithreaded programs is challenging, as assertion evaluation should not interfere with the execution of other threads. This paper describes the prototype implementation of a run-time assertion checker that achieves this by evaluating assertions over snapshots of the state, instead of over the live state. Our prototype e-OpenJML, an extension to OpenJML, provides an easy to use, safe and interference-free evaluation of JML specifications in multithreaded programs. To achieve this, it integrates e-STROBE, our extension to the STROBE framework for asynchronous assertion evaluation. e-STROBE prevents all possible interferences between assertion evaluation and other program threads, which the original STROBE can not. It also simplifies evaluating assertions that relate the value of expressions in multiple states.

1. INTRODUCTION

Run-time assertion checking is an attractive, easy to use technique which integrates formal analysis with system development. To enable run-time assertion checking, a programmer only has to annotate a program with assertions, expressing the properties that he expects to hold at a particular point in the program. Then, whenever program execution reaches such an assertion, it will be evaluated, and if it does not hold the program will terminate and indicate the violation. One popular application of this approach is extended testing, whereby intermediate states during the execution of a tested operation are considered in addition to just its output. A second example is security monitoring [21] with security properties encoded as assertions.

For the implementation of a run-time assertion checker, several crucial properties have to be considered [7]: (*i*) checking should be *transparent*, *i.e.*, if there are no assertion failures, program behaviour should be the same when the program is executed with assertion checking enabled or disabled; (*ii*) it should be possible to *isolate* the problem, *i.e.*, whenever an assertion fails, it should report the assertion failure at that point in the execution, and not at an arbitrary later point; and (*iii*) the checker should be *trustworthy*, *i.e.*, if a failure is signaled, there indeed should be one. Most run-time assertion checkers do not consider multithreading explicitly, and in particular they do not prevent interference between evaluation of an assertion and program code. As a consequence, *existing run-time assertion checkers are not trustworthy in the context of multithreaded programs.*

This paper describes *e-OpenJML*, a run-time assertion checker for JML that is transparent, isolating and trustworthy also in the context of multithreaded programs. E-OpenJML is built using *e-STROBE*, an extension of the STROBE framework [1] for asynchronous assertion checking, which evaluates assertions over snapshots, *i.e.*, copies of program states. Snapshot usage ensures that assertions are evaluated in a stable state, which cannot be changed by other threads, thus making assertion evaluation interference-free. To provide support for a high-level annotation language, e-OpenJML integrates OpenJML with e-STROBE to enable interference-free assertion checking of JML.

More precisely, this paper contributes the following:

- e-STROBE, an extension of the STROBE framework that makes it possible to check assertions completely interference-free, and to check assertions that express relations over multiple states;
- e-OpenJML, an integration of e-STROBE and OpenJML, which provides safe assertion checking of multithreaded programs without any false positives and false negatives; and
- validation of our implementation in terms of demonstrators forcing threads to be scheduled such that naive assertion checking is guaranteed to generate either a false positive or false negative, while interference-free assertion checking correctly evaluates the assertions.

2. A QUICK OVERVIEW OF JML

JML is a behavioural interface specification language for Java [18]. Its specifications of the intended behaviour of methods and classes are written as Java boolean expressions, enriched with a few specification-specific constructs, making them look familiar to Java programmers.

2.1 The JML Specification Language

The basic specification constructs in JML are pre- and postconditions of methods (specified by keywords **requires** and **ensures**, respectively). Properties that should be preserved throughout the lifetime of an object are specified

*Partially supported by ERC grant 258405 (VerCors project).

FTfJP'17th Workshop on Formal Techniques for Java-like Programs, July 07 2015, Prague, Czech Republic
Copyright 2015 ACM ISBN 978-1-4503-3656-7/15/07
http://dx.doi.org/10.1145/2786536.2786541 ...$15.00.

as class invariants (keyword **invariant**). History constraints specify a relation between a visible state and any visible state that occurs later in the program (keyword **constraint**). In addition to boolean Java expressions, specifications can use a few extra constructs: \result denotes the result of a method call in the postcondition, \old(E) indicates that an expression E should be evaluated in the pre-state of a method, \forall, \exists, ==> *etc.* can be used to build up logical expressions.

All JML specifications are written as special comments, starting with //@ or /*@. This ensures that JML specifications are simply ignored by a normal compiler, while all tools supporting JML can recognize the specifications.

2.2 OpenJML: Tool Support for JML

One of the attractive features of JML is that it has a large variety of tool support. This paper focuses on run-time checking; however there also exists support for, *e.g.*, static checking, annotation generation, and test case generation (see [7] and `http://www.jmlspecs.org`). One of the most recent, and still actively developed tools is OpenJML [10,11], which provides support both for run-time and static checking. Run-time checking in OpenJML is implemented by translating each high-level specification construct into assertions that capture these specifications. For example, method pre- and postconditions are translated into assertions before and after the method body. Whenever an \old expression is used in a postcondition, OpenJML pre-calculates the value of the expression in the pre-state and stores this in a local variable, which is then used in the postcondition. Class invariants are checked as part of the pre- and postconditions of each method in this class (or sub-classes thereof); history constraints are added to a method's postcondition.

2.3 Motivating Example

OpenJML's support for run-time checking does not make any explicit considerations about multithreading, thus evaluations of assertions might be influenced by other threads. Consider for example the class DisplaySet in Listing 1.

```
class DisplaySet {
  Display d1; Display d2;

  //@ requires t != null;
  //@ ensures d1.getTime() == d2.getTime();
  DisplaySet(Timer t){
    d1 = new Display(t); d2 = new Display(t); }

  public static void main(String[] args){
    Timer t = new Timer();
    t.start();
    DisplaySet d = new DisplaySet(t);
} }
class Display {
  Timer timer;

  Display(Timer timer){
    this.timer = timer; }

  /*@ pure */ int getTime(){
    return timer.getTime();
} }
```

Listing 1: Example JML specification that can yield false positive/negatives in a multi-threaded context

In the postcondition of DisplaySet's constructor, getTime is invoked twice. Since it is called twice on the same object, one would expect these two calls to return the same value. However, in a multithreaded setting, another thread could update the value of timer in between the two calls, resulting in an assertion failure, while in fact there is no problem, as specification evaluation should be atomic. Similarly, assertions might also *not* fail because of interference. Our goal is to develop a run-time checker that avoids such *false positives and negatives.*

3. THE STROBE FRAMEWORK

The STROBE framework [1] provides support for efficient run-time checking by means of *asynchronous assertion checking*, *i.e.*, assertions are evaluated in a separate *checker thread*, while the regular *program threads* continue their normal execution. To make sure that assertions are evaluated in the correct state, *i.e.*, the program state at the moment when evaluation of the assertion started, the STROBE framework uses snapshots. In e-STROBE, our extension of STROBE described in Section 4, we use snapshots to guarantee that assertion evaluation is not affected by interference. This section describes the essential ingredients of the existing STROBE framework, as well as the limitations that we had to address to adapt e-STROBE for interference-free run-time assertion checking of multithreaded programs.

3.1 Implementation

The STROBE framework is developed on top of the Jikes-RVM [2], a Java Virtual Machine designed for easy experimentation with new language designs and implementations. Its technology and performance are comparable to those of commercial JVMs, and thus the STROBE approach (including our extensions) is also applicable to other JVMs.

Asynchronous evaluation of assertions in STROBE is implemented using *futures* [15, 17]. Assertions are evaluated by returning a handle to a future object, which encodes a computation whose evaluation will denote validity of the assertion in this execution. The future is evaluated in a separate checker thread. If the program at some point requires the result of the future computation, it will block until this result becomes available, if it is not already. This ensures that run-time assertion evaluation only has limited impact on the performance of normal program execution.

Evaluation of assertions in a separate thread is *safe*, *i.e.*, it does not change program behaviour, if the following two conditions are respected[1]: (*i*) the assertion should be *pure*, *i.e.*, it should never alter the program state; and (*ii*) evaluation of the assertion should not see any updates made by the regular program threads *after* the future was created.

To ensure that assertions do not see any later updates made by the regular program threads, the future is evaluated in a *snapshot state.* Conceptually, the snapshot is a frozen copy of the normal *live* program state, in which the related assertion (the future) has been created. We say that in this state the snapshot has been *initialised* and becomes *active.* It stays active until evaluation of the future is complete.

The original program is unaware of the use of snapshots. Object references in snapshots are evaluated as normal object references, and might actually point into the live state

[1]This is a simplified formulation of the general safety rules for futures [22], restricted to the specific case of assertions.

again. It is the STROBE framework that ensures that object accesses are transparently rerouted to the snapshot state.

To create the snapshots, simply copying the full program state for assertion evaluation every time a future is created would have serious impact on program performance. Instead, STROBE uses a *copy-on-write* strategy, only making copies of that part of the program state that is actually changed. STROBE also supports multiple snapshots, allowing multiple asynchronous assertions to be evaluated in parallel on snapshots of different versions of the live states.

References to all copies of a given object are stored in the object header in a *forwarding array.* The array has as many entries as active snapshots can exist, and if an object has a copy for a specific snapshot, a reference to this copy is stored in the array entry corresponding to this snapshot. In STROBE a checker thread is always associated with one active snapshot. As STROBE uses a fixed number of checker threads, the maximum number of active snapshots, and thus also the size of the forwarding array is fixed.

When an object o is modified then in all entries of o's forwarding array that correspond to an active snapshot and that are still empty, a new copy of o is stored. For this purpose, the STROBE framework uses *substituting write barriers*, as provided by JikesRVM [5]. These special code blocks are executed in place of regular write operations and perform additional actions like maintaining the forwarding array. Similarly, read barriers ensure that the right object copy is used, depending on whether the live state or a specific snapshot is currently active.[2] For technical reasons, STROBE requires methods containing asynchronous assertions, *i.e.* those where read barriers need to be used, to be annotated with a @ConcurrentCheck annotation.

3.2 Limitations

Some of the limitations of STROBE are implementation issues, and have work-arounds, *e.g.*, STROBE has no support for snapshots of variables with native types or for static variables, and snapshot states can only be used in methods annotated with @ConcurrentCheck, which hinders assertion checking in library classes that cannot be modified.

However, there are also some conceptual limitations of STROBE, that we had to address while developing e-STROBE. Firstly, evaluating assertions that *relate behaviour in two different states* (*e.g.*, relating the pre-state and the post-state of a method) is complicated in STROBE. When supporting higher-level annotations, such as postconditions and history constraints, efficient support for assertions over multiple states is essential. In STROBE this is achieved by storing expression evaluations over the first state in global variables, and then passing these global variables to the second state. These global variables are also visible outside snapshots, which violates one of the basic rules of the STROBE framework.

Secondly and most importantly, the STROBE framework does not prevent annotation interference. Consider method equalsTest in Listing 2, which creates an assertion (lines 7–9), starts the evaluation of the future (line 10), waits for the future and asserts that the result is *true* (line 11). The future will evaluate an instance of EqualsAssert (line 8) over a snapshot state. However the argument values of this assertion are determined over the live state (line 9), which is vulnerable to interference, as a second program thread could modify the Node.next variable right between the two calls to getNext, which makes the assertion fail.

```
1   public class Node {
2     private Node next;
3
4     private Node getNext(){ return next; }
5
6     public void equalsTest(){
7       CHAFuture equalsFuture =
8         new CHAFuture(new EqualsAssert(
9           this.getNext(), this.getNext()));
10      equalsFuture.go();
11      assert equalsFuture.get();
12  } }
```

Listing 2: Assertion interference in STROBE

4. STROBE FOR MULTIPLE THREADS

To address this problem presented above, we have adapted STROBE and changed how assertion evaluation is initialised. Concretely, in e-STROBE, snapshot management is exposed to the programmer, giving the programmer full control over the state on which an expression is evaluated. This has the advantage that initialization of the assertion is done directly on the snapshot, which guarantees that assertion evaluation is completely free of interference. Revisiting the example in Listing 2, in e-STROBE the snapshot is initialised at the beginning of equalsTest, such that the two calls to getNext on line 9 are evaluated over the same snapshot state, ignoring potential concurrent changes in the live state.

For our prototype, it is most important that assertions are evaluated interference-free, and performance is less of an issue. Therefore, in e-STROBE, we do not use futures, unlike the STROBE framework, but we focus on synchronous evaluation of assertions. Nevertheless, our techniques also could be applied when assertions are evaluated asynchronously.

4.1 Snapshots in E-STROBE

In STROBE, snapshots are set up in such a way that they can only be used by checker threads. In contrast, e-STROBE provides several methods for managing snapshots in *program threads.* The method initiateProbe initialises a new snapshot and returns a corresponding identifier. This can be passed to completeProbe to destroy the snapshot. We have also added two methods to the JikesRVM class RVMThread: switchToSnapshot forces the thread to execute in the specified snapshot's state; and switchToLive forces the thread to execute in the live state. Notice that snapshot identifiers might be shared by threads. The method switchToSnapshot ensures that the specified snapshot is *valid*, *i.e.*, it has been initialised and has not been destroyed yet. A program should explicitly destroy snapshots. If this is forgotten, it will not affect correctness of the program, but it might have an impact on performance, because the snapshots will keep on growing as the program state further evolves.

Listing 3 gives a concrete example how e-STROBE supports manipulation of snapshots. The statements at lines 15 and 17 are evaluated over snapshots, initialised at lines 11

[2]The extra workload of the write and read barriers slightly modifies the program's timing compared to a run without STROBE. This potentially affects the program's non-deterministic behavior but does not impair the results of assertion checking, which is valid for the program run under scrutiny.

and 13, respectively. A thread can initialise multiple snapshots, but it can evaluate an expression over one snapshot at a time only. After initialization of the second snapshot in line 13, any change in the program state is recorded in both snapshots (with identifiers preId and postId, respectively).

The explicit switching between snapshots and the live state (lines 14, 16 and 18) allows us to identify which regions of the program are evaluated in a snapshot state. Such regions are said to be within the *scope* of the associated snapshot. All other statements are in the *live scope*.

```
 1  class Timer extends Thread {
 2    Integer time = 0;
 3
 4    @ConcurrentCheck
 5    int getTime(){ return time; }
 6
 7    @ConcurrentCheck
 8    void increase(){
 9      RVMThread currentThread =
10        RVMThread.getCurrentThread();
11      int preId = Snapshot.initiateProbe();
12      time = time + 1;
13      int postId = Snapshot.initiateProbe();
14      currentThread.switchToSnapshot(preId);
15      Integer oldTime = this.getTime();
16      currentThread.switchToSnapshot(postId);
17      assert this.getTime() > oldTime;
18      currentThread.switchToLive();
19      Snapshot.completeProbe(preId);
20      Snapshot.completeProbe(postId);
21  } }
```

Listing 3: E-STROBE's manipulation of snapshots

A program thread executing on a snapshot should not change any objects in the snapshot state, as these changes would be visible in the live state of the program as well. The e-STROBE framework currently does not explicitly check for this, as we concentrate on its use for assertion evaluation, and assertions should always be free of side-effects. Moreover, as we use e-STROBE to check specifications in a higher-level specification language such as JML, type checking will ensure that the assertions are indeed free of side effects. However, note that it *is* allowed to create new, temporary objects in a snapshot scope. According to, *e.g.*, JML [12], object creation does not impede the purity of assertion evaluation on a snapshot state.

4.2 E-STROBE Implementation

In e-STROBE, we follow the STROBE implementation of snapshots. However, as e-STROBE snapshots are not tied to checker threads, the number of possible snapshots to maintain is much higher. While in principle this number is unbounded, we specify a fixed (but arbitrary) maximum[3]. As we intend to use snapshots for assertion checking, the number of snapshots needed depends on the number of concurrent threads and the number of snapshots that are kept for evaluating multiple state assertions. This maximum can be set upon compilation. In our experiments, we used 1000, which is certainly enough for regular desktop applications.

When a snapshot is initialised, it triggers e-STROBE to incrementally preserve the program state at the moment of snapshot initialization. For the preservation of the program state, the STROBE technology is reused without changes. Initialization of a snapshot is atomic and transparent: it will not change the behaviour of program fragments that are evaluated outside the scope of the newly initialised snapshot.

When a snapshot is initialised, it always takes the live state as its starting point. Thus even when snapshot A is initialised within the scope of snapshot B, it will be aware of all changes that were made after initialization of snapshot B.

When a snapshot is destroyed, the program stops the incremental preservation of the program state. All object copies that were preserved for the snapshot may be cleaned up by the garbage collector, unless they are still in use for another snapshot. Snapshot destruction is—just like initialisation—atomic and transparent. The current implementation of e-STROBE checks if a program tries to switch to a snapshot that does not exist, but it does not check if a snapshot is still in use when it is destroyed. For now, this is the responsibility of the programmer; however in the future we would like to embed such checks in the system.

To enforce that objects are read in the appropriate snapshot state, and that snapshots are properly maintained, e-STROBE uses barriers much in the same way as STROBE, with the following differences: (*i*) all checks are based on the snapshot identifier, whereas STROBE used the checker thread identifier; and (*ii*) the read barriers within the methods annotated with the @ConcurrentCheck annotation are enabled for *all* threads in the program, whereas in STROBE they are only enabled for checker threads.

4.3 Validation

To validate that assertion checking is indeed interference-free, we create a program (in Listing 4) that synchronizes two threads in such a way that assertion interference *must* occur in a *normal* execution, *i.e.*, when the assertion and the program threads use the same live program state. This can *not* occur if we execute the program using e-STROBE. For this demonstration, we use assertions with side effects, which of course normally should not be done.

```
public class Timer extends Thread {
  Integer time = 0;

  @ConcurrentCheck
  int getTime(){ return time; }

  @Override
  public void run(){
    this.time++;
    this.time++;
} }

public class Display {
  Timer timer;

  Display(Timer timer){ this.timer = timer; }

  @ConcurrentCheck
  int getTime(){ return timer.getTime(); }

  @ConcurrentCheck
  static void testDisplay(){
    Timer t = new Timer();
    t.start();
    Display d1 = new Display(t);
```

[3]Making the forwarding array grow dynamically based on the actual number of snapshots is not feasible, as it would require updating this array for all existing objects.

```
26        Display d2 = new Display(t);
27        RVMThread currentThread =
28            RVMThread.getCurrentThread();
29        int snapshotId = Snapshot.initiateProbe();
30        currentThread.switchToSnapshot(snapshotId);
31        assert d1.getTime() == d2.getTime();
32        currentThread.switchToLive();
33        Snapshot.completeProbe(snapshotId);
34    } }
```

Listing 4: Simplified code used for validation

Using thread synchronization primitives not shown in the example code, we ensure an execution with the following thread interleaving:

	Main thread	Timer thread	t.time (live)
1	d1.getTime() (l. 31)		0
2		t.time++ (l. 9)	1
3	**return** t.time (l. 5)		1
4	d2.getTime() (l. 31)		1
5		t.time++ (l. 10)	2
6	**return** t.time (l. 5)		2

Thus, if the assertion (line 31) is evaluated over the live state, the calls to d1.getTime() and d2.getTime() return different values (1 and 2 respectively). However, if we wrap the assertion by snapshot management code, the assertion is evaluated over the snapshot, and both calls to getTime() yield 0, *i.e.*, the timer's value at snapshot initialization time.

5. OPENJML FOR MULTIPLE THREADS

We integrated the use of snapshots in the runtime assertion checker of OpenJML, to enable interference-free checking of assertions for multithreaded programs. This allows application developers to write JML specifications, without having to worry about manipulating snapshots, as this is all taken care of in our OpenJML extension, called e-OpenJML. Notice that this integration is only a proof-of-concept and focuses on pre- and postcondition checking (including class invariants and history constraints), *i.e.*, the core of JML specifications. However, it demonstrates that the approach works without changing how the user writes specifications.

We have adapted the OpenJML implementation so that pre- and postconditions are evaluated over snapshots, instead of over the live state. To illustrate the code generation strategy, Listing 5 shows an example of a method compiled with e-OpenJML. We use e-STROBE to make a snapshot for the pre-state (line 4) and the post-state (line 13) of the method. Preconditions and **old** expressions from the postcondition are evaluated over the pre-state snapshot. For this purpose, the corresponding statements (line 6–7) are wrapped between statements switching from the live state to the pre-state snapshot and back (lines 5 and 8). Postconditions are evaluated in the same way over the post-state snapshot (lines 14–18). Any **old** expression is evaluated before the original method body on the pre-state snapshot and the result is stored in a local variable (line 6), which is used in the post-state snapshot (line 16). At the end of the code that builds the checks for pre- and postconditions, the snapshots are destroyed (lines 9 and 19).

```
1   public void addNode(Node node) {
2       RVMThread currentThread =
3           RVMThread.getCurrentThread();
4       int preId = Snapshot.initiateProbe();
5       currentThread.switchToSnapshot(preId);
6       int oldLength = _JML$model$length();
7       assert !_JML_METHODEF__contains(node);
8       currentThread.switchToLive();
9       Snapshot.completeProbe(preId);
10      node.next = this.next;
11      this.next = node;
12      calculatedLength = calculatedLength + 1;
13      int postId = Snapshot.initiateProbe();
14      currentThread.switchToSnapshot(postId;
15      assert _JML_METHODEF__contains(node);
16      assert _JML$model$length() == oldLength + 1;
17      assert calculatedLength == _JML$model$length();
18      currentThread.switchToLive();
19      Snapshot.completeProbe(postId);
20  }
```

Listing 5: Code with snapshot management generated by e-OpenJML from an annotation

The e-OpenJML implementation changes the contextual analysis phase of the OpenJML compiler, where extra statements are inserted to evaluate the pre- and postconditions of methods. These statements may contain calls to other methods generated by OpenJML, which may either be called from within or from without a snapshot scope and thus, they may themselves initialise and use snapshots. However, in e-STROBE such a second snapshot would be initialised from the live state. Therefore, the called method would operate over a program state that is different from the snapshot used to evaluate the assertions initially. To ensure that all parts of the assertions are evaluated over the same snapshot, we add some extra run-time utility methods that check whether a snapshot scope is currently active and only initialise a snapshot if this is not the case.

The current implementation of e-OpenJML only works for methods that contain a @ConcurrentCheck annotation. While this annotation potentially can be added automatically by e-OpenJML, the current implementation requires the user to add this annotation manually.

In a similar way as we validated that e-STROBE provided interference-free assertion checking, we also validated e-OpenJML by using a JML-annotated version of classes Timer and Display, available online[4]. When checked with the original OpenJML tool, an assertion violation will be falsely reported, while when checking with e-OpenJML, the assertion violation is not reported anymore, because the assertion is evaluated over the snapshot.

6. RELATED WORK

Several implementations of run-time checkers for specification languages such as JML exist, *e.g.*, OpenJML [11], the old JML2 tool set [8,9], AspectJML [19], and SAGA [13, 14]. However, these implementations do not consider multi-threading explicitly, and they do not provide any precautions against assertion interference.

Most work that is specific to assertion evaluation of concurrent programs, is related to static verification, using (extended) permission-based separation logic, see e.g., [3, 6]. There also exists a proposal to extend JML with several constructs to capture the behaviour of locks and other non-

[4]https://github.com/Scharrels/e−openjml/

interference properties as part of the specifications, known as JML level C [20]. This does not provide any special support for preventing assertion interference to occur, though. Moreover, we are not aware of any tool support for JML level C.

Aurujo et al. [4] present a technique for interference-free run-time checking by using safe-points, which are specific points where the program itself must guarantee protection against assertion interference. A program should acquire the locks that it needs to ensure the freedom of interference before it enters this safe point. Annotations can then be checked at safe points only.

7. CONCLUSIONS AND FUTURE WORK

This paper presents an extension of the STROBE framework that makes runtime assertion checking in multithreaded applications completely interference-free, by allowing program threads to manipulate snapshots of the program state directly. To hide snapshot manipulation from the programmer, our e-STROBE extension is integrated with OpenJML. Thus, an application developer can specify the desired behaviour as JML specifications, and our e-OpenJML system internally uses the e-STROBE framework to ensure that the assertions capturing the JML specifications are checked in an interference-free manner. E-STROBE and e-OpenJML are available for download[5].

What is more, our approach also facilitates a new, more expressive way of evaluating assertions that relate multiple states. As an example, consider a postcondition where the **\result** keyword is used inside an **\old** expression, *e.g.*, assume a is an array of Timer objects and we write a postcondition for a method that should first increment a timer stored in a, and then return an index to this timer. We can specify this in JML as: a[**\result**].getTime() == **\old**(a[**\result**]. getTime()) + 1. However, OpenJML does not allow the use of **\result** in **\old** expressions. With our approach, we can handle this specific case by saving the entire array a in the pre-state, calculating the array index in the post-state, and evaluating the expression (*i.e.*, getting the time of the selected Timer object) in the pre-state again. Thus, in the evaluation of postconditions, we can mix values calculated in the post- *and* in the pre-state of the method. In this particular example, we can see the unchanged state of the Timer object, independent of whether it has been modified during method execution.

To increase the usefulness of run-time checking, it is important to analyse different executions. For sequential programs, this is achieved by varying the input parameters. However, for multithreaded programs, also the scheduling of the different threads should be taken into account. This cannot be steered by changing input parameters alone. Therefore, we plan to look into ways to control the scheduler, for example using a technique like CalFuzzer [16].

Finally, so far, we only looked at interference-free checking of assertions expressing properties of data. However, most work on the static verification of concurrent programs focuses on absence of data races, and uses a form of permissions to guarantee freedom of interference [3, 6]. We plan to study run-time checking of permission-based specifications, which requires that permissions are traced during execution. If permissions are used, it means that the problem of assertion interference disappears: if a specification is properly framed by permissions, it is interference-free. However, using permissions is a more advanced specification technique, and for practical purposes and quick feedback, our current implementation will still be useful.

[5]https://github.com/Scharrels/e−openjml/

8. REFERENCES

[1] E. E. Aftandilian, S. Z. Guyer, M. Vechev, and E. Yahav. Asynchronous assertions. In *OOPSLA '11*, pages 275–288. ACM Press, 2011.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[3] A. Amighi, S. Blom, S. Darabi, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski. Verification of concurrent systems with VerCors. In *SFM-14:ESM*, volume 8483 of *LNCS*, pages 172–216. Springer, 2014.

[4] W. Araujo, L. C. Briand, and Y. Labiche. Enabling the runtime assertion checking of concurrent contracts for the Java modeling language. In *ICSE '11*, page 786. ACM Press, 2011.

[5] S. Blackburn, R. Garner, and D. Frampton. MMTk: The Memory Management Toolkit, 2006.

[6] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL'05*, pages 259–270. ACM, 2005.

[7] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, June 2005.

[8] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Department of Computer Science, Iowa State University, Ames, 2003.

[9] Y. Cheon and G. T. Leavens. A Runtime Assertion Checker for the Java Modeling Language. In *International Conference on Software Engineering Research and Practice*, pages 322–328, 2002.

[10] D. R. Cok. OpenJML: JML for Java 7 by Extending OpenJDK. In *NFM'11*, pages 472–479, Berlin, Heidelberg, 2011. Springer.

[11] D. R. Cok. OpenJML User Guide, 2013.

[12] D. R. Cok and G. T. Leavens. Extensions of the theory of observational purity and a practical design for JML. In *SAVCBS 2008*, pages 43–50, 2008.

[13] C. P. T. de Gouw and F. S. de Boer. Run-time verification of black-box components using behavioral specifications: An experience report on tool development. In *FACS2012*, volume 7684 of *LNCS*, pages 128 – 133. Springer, 2013.

[14] C. P. T. de Gouw, F. S. de Boer, and J. J. Vinju. Prototyping a tool environment for run-time assertion checking in JML with communication histories. In *FTfJP 2010*. ACM, 2010.

[15] R. H. Halstead, Jr. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, Oct. 1985.

[16] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An extensible active testing framework for concurrent programs. In *CAV '09*, pages 675–681. Springer, 2009.

[17] D. Lea. JSR166: Concurrency Utilities. https://www.jcp.org/en/jsr/detail?id=166, 2004.

[18] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw Eng Notes*, 31(3):1–38, 2006.

[19] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. Zimmerman, M. Cornélio, and T. Thüm. AspectJML: Modular specification and runtime checking for crosscutting contracts. In *Modularity 2014*, 2014.

[20] E. Rodriguez, M. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *ECOOP 2005*, pages 551–576. Springer, 2005.

[21] F. Schneider. Enforceable security policies. Technical Report TR99-1759, Cornell University, October 1999.

[22] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *ACM SIGPLAN Notices*, volume 40, pages 439–453. ACM, 2005.