UNIVERSITY OF TWENTE.

Master Thesis

Runtime assertion checking of multithreaded Java programs

An extension of the STROBE framework

Author:
Jorne Kandziora

Supervisors:
dr. M. Huisman
dr. C.M. Bockisch
M. Zaharieva-Stojanovski, MSc

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in the

Formal Methods and Tools group,
Department of Computer Science,
Faculty of Electrical Engineering, Mathematics and Computer Science.

August 29, 2014

Abstract

It is important that software is correct, and there are many different techniques to guarantee the correctness of a program. One such technique is runtime assertion checking: a technique that evaluates properties of a program during execution. This particular technique scales well, and can therefore be used to check the behaviour of large pieces of software.

Many runtime assertion checkers are only built for sequential programs and do not guarantee correct behaviour if they are used in a multithreaded environment. We will show that runtime assertion checkers in a multithreaded environment can sometimes report errors that do not exist in the program or fail to detect problems that exist, both of which is problematic.

To solve this problem, we build an extension to the STROBE framework. STROBE is built to accelerate runtime assertion checking with the use of asynchronous assertions: checks that are executed in parallel to the program itself. This framework performs this checking correctly with the use of snapshots, that efficiently save the previous states of a program. We extend the STROBE framework into the e-STROBE framework, that uses these snapshots in a multithreaded program, to protect the program from unwanted interference during runtime assertion checking.

We apply the snapshots in the e-STROBE framework to OpenJML, to show that existing runtime assertion checkers can be modified to work correctly in a multithreaded environment. We will give several examples to show that our solution works, and gives the expected results in both single-threaded and multithreaded programs.

Contents

Chapter 1

Introduction

We want our software to behave correctly, but it is often hard to describe what correct behaviour of a program exactly means, and when a program behaves as it should. Therefore, many techniques have been designed to describe, check and verify correct behaviour, including unit testing, static checking and the technique that we will explore in this thesis: runtime assertion checking. All these techniques have their different uses: unit testing tests the different modules of a program with a pre-defined set of test cases, static checking proves that the software behaves correctly for all possible inputs and runtime assertion checking tests the program during a specific execution.

All these techniques require a description of the correct behaviour of a program, and provide a means to check if the program behaves in a way that meets this description. We will call such a description the *specification* of a program.

1.1 Runtime assertion checking

A simple way to check if the program executes correctly according to its specification is by adding extra code to a program to check this. This technique is called *runtime assertion checking*. Programming languages often have some native support for assertion checking. Furthermore, standalone systems exist to aid the user in the writing and checking of assertions.

Runtime assertion checking only checks the correct behaviour of a program during a particular execution of the program, and, unlike static checking, does not guarantee the overall correctness of the program. Therefore, a different execution of the same program might not behave correctly according to its specification. A runtime assertion checker will only detect such an error if the incorrect behaviour actually occurs during an execution of the program.

However, runtime assertion checking is scalable and easy to implement. Therefore, this technique is widely used, and can quickly be used to test large pieces of software.

1.1.1 OpenJML

In this thesis, we will demonstrate runtime assertion checking for Java programs using a standalone runtime assertion checker, called OpenJML [16]. This tool processes specifications that are written in a separate specification language, called the Java Modeling Language (JML) [22]. OpenJML can translate such a specification into the necessary statements to check if the program executes correctly according to its specification, and automatically adds these statements to the program.

1.2 Multithreaded programs

Nowadays, programs no longer execute one task at the time, but run several independent tasks concurrently. Programs that do so, are called *multithreaded programs*. However, tasks that are executed in parallel can interfere with each other. Therefore, incorrect behaviour of a program is no longer solely caused by the task that is executed, but can also be caused by the incorrect behaviour of another task of the program.

1.2.1 Runtime assertion checking in multithreaded programs

The specification of a program changes if we have to account for multiple threads that influence each other. We can no longer just check the current state of the program, but we have to account for all ways in which the program can influence a specific task of the program. However, the principles of runtime assertion checking remain mostly the same: instead of executing checks during the execution of one task, we now have to insert checks for all the tasks that we perform, and check that all tasks have thus far behaved correctly. A runtime checker will still only check a particular execution of the program.

A multithreaded program can also influence the results of the runtime assertion checker code itself. This can lead to a situation where the runtime assertion checker does not detect a problem that it should have detected, or reports a problem that is not present in the program. If this happens, we can no longer trust the results of the runtime assertion checker: the reporting of a problem does not necessarily mean that the behaviour of the program is incorrect. Likewise, interference during runtime checking might cause the runtime assertion checker to miss an error in the program, that could have been detected. We will show an example of both in Section 4.2 and 5.4.

1.2.2 Asynchronous assertions in the STROBE framework

Runtime assertion checking is performed during the execution of the program, and thus slows down the program. The STROBE framework [7] uses an alternative checking mechanism called asynchronous assertions, that executes these checks in parallel, minimizing the time that is spent on assertion checking.

Asynchronous assertions are executed as independent tasks in a program, and the outcome of the checks can thus be influenced by the program itself when it continues its execution. The STROBE framework prevents this from happening by executing asynchronous assertions against a copy of the memory, called a *snapshot*. These snapshots ensure that an assertion is evaluated against a state of the memory that does not change during the evaluation. In this way, the asynchronous assertion can safely check the behaviour of the program, while the execution itself continues.

The STROBE framework focuses on the efficient execution of checks. The behaviour of asynchronous assertions is primarily checked for the execution of single-threaded programs. However, the framework can be used to perform checks in multithreaded programs. We will examine the behaviour of asynchronous assertions in such programs in Chapter 3.

1.3 Contribution

Snapshots are an efficient method to ensure the correct execution of checks in a multithreaded program. We will therefore present the e-STROBE framework, as an extension to the STROBE

framework. The e-STROBE framework will not only support snapshots in asynchronous assertions, but will also provide an interface to use snapshots during the synchronous assertion checking of programs. Supporting synchronous assertions will ease the integration of the e-STROBE framework in existing runtime assertion checking.

We will show that the e-STROBE framework can be used to write checks in multithreaded programs, and execute them safely against a copy of the memory. We will also show how the checks that are performed with snapshots can be moved around, allowing a runtime assertion checker to perform checks on state transitions that could previously not be checked. We will provide examples of snapshots in both single-threaded and multithreaded programs, and test that the e-STROBE framework provides the specified protection to assertions in Java.

We will show that some of the checks in OpenJML can not be trusted in multithreaded programs. We will provide a solution to this problem using the snapshots in the e-STROBE framework, and modify OpenJML in such a way that the checks that are performed by this runtime assertion checker automatically use snapshots. We will give examples of single-threaded and multithreaded programs in OpenJML, and show how the e-STROBE framework is integrated in OpenJML to ensure the correct behaviour of checks in these examples. We will test the provided solution, both through decompilation and by means of a multithreaded program that will deterministically trigger failures in assertions if the e-STROBE framework is not used in OpenJML.

1.4 Outline

The work that is performed in this project consists of two main contributions: the extension of the STROBE framework and the modification of OpenJML. Both contributions require an introduction to the original piece of software, followed by a discussion of the changes that are made in this project. Before we discuss these tools, we will give an exact description of the problem at hand, and provide the necessary terminology in Chapter 2.

Chapter 3 will introduce the STROBE framework. We will explain how this framework implements assertion checking, and show that there are some limitations to the use of the STROBE framework in multithreaded programs.

We will introduce the e-STROBE framework in Chapter 4. We will describe a snapshot interface that can be used without asynchronous assertions, and show how this interface can be used to overcome the limitations that we discussed in Chapter 3.

Chapter 5 will discuss the current state of OpenJML. We will describe the use of the Java Modeling Language to create specifications, and explain how OpenJML uses these specifications to create checks for a program. Finally, we will show how OpenJML checks in a multithreaded program can sometimes have an unexpected result.

Chapter 6 discusses our changes to the OpenJML runtime assertion checker, allowing OpenJML to use the e-STROBE framework to ensure the expected result of its checks in multithreaded programs. We will give examples of this integration, and test that our implementation behaves as intended.

Chapter 7 will give a summary of the results of this thesis, and evaluates the choices that were made. We will discuss the limitations of the provided solutions, and suggest possible improvements and future work.

Chapter 2

Problem analysis

This chapter will give a detailed analysis of the problem that is solved in this thesis. It explains the different programming concepts that are used and introduces the terminology that is used in the rest of the thesis. We will end this chapter with the research question, that defines the goal for this research.

2.1 Faults and failures

Programming a piece of software will seldomly be flawless. *Faults* will be introduced during the programming of software: program statements that contribute to the incorrect behaviour of a program. If these faults are not found during the testing of a program, they will eventually lead to a *failure*: undesired behaviour of the program, visible to the user. A faulty program often runs for a long time before a failure manifests. This makes it hard to detect the fault that eventually leads to the failure of the program.

2.2 Assertions

One way to pinpoint the location of a fault in a program is to annotate it with expectations about the state at a certain point in the program. Satisfying these expectations increases the confidence in the correctness of the program up to that point. Expectations can be written in the form of *assertions*: boolean expressions that we expect to be true at a certain point in the program.

Assertions are originally designed as a means to reason about the correctness of the program. [19, 28] The original proposals were to use assertions as a form of documentation and to aid in the construction of a deductive proof of the correctness of a program. Modern languages have often incorporated assertions in the language, to monitor the correct behaviour of the program during execution. Java uses the **assert** statement for this purpose.

If a language lacks the support, or the expressiveness needed to express assertions, a *preprocessor* or alternative compiler can be used to translate assertions into executable program code. The complete system that is responsible for the compilation, execution and monitoring of assertions is called a *runtime assertion checker*.

2.2.1 Specification of a program

Programs are structured in *blocks* of statements, that are built to perform a certain task, e.g. executing sequentially or conditionally using respectively **if** and **while** statements. Many programming languages offer some form of procedures: blocks of statements that can be invoked repetitively. Object-oriented languages offer *methods* and *classes* to add even more structure to a program. Assertions are often used to reason about the task performed by a block of statements.

Assertions that are used to reason about the state of a program before a block of code is executed are called *preconditions*. Assertions that reason about the state of a program after the execution of a block are called *postconditions*. When an assertion should hold in every state of a program, we call this assertion an *invariant* of the program. *Loop invariants* are used to describe properties that should hold before and after every iteration of the loop body and *class invariants* hold whenever the instance of a class is used in the program.

We can use assertions to construct a full description of the behaviour of an entire program. Such a description is called the *specification* of a program. Specifying a program using preconditions, postconditions and invariants is better known as the *Design by Contract* method [23]. This method was originally developed for the Eiffel programming language and later adopted by other programming languages and runtime assertion checkers.

2.2.2 Assertion violations

When an assertion is false during a certain execution of a program, we call this an *assertion violation*. The reporting of an assertion violation by a runtime assertion checker will often be caused by code that behaves incorrectly according to its specification. Sometimes, the specification itself is incorrect. We call an assertion violation that is caused by an incorrect specification a *false positive*.

A badly written runtime assertion checker might incorrectly report an assertion violation assertion while the assertion itself is true. The programmer cannot solve such a problem, as the program behaves correctly according to its specification. Burdy et al. [14] define that we expect a runtime assertion checker to be *trustworthy*: "every assertion violation must be a report of an assertion that is false."

2.2.3 Transparency

Burdy et al. also state that a runtime assertion checker should be *transparent*: the execution of a program without assertions must behave exactly the same as the execution of the same program with assertions enabled, apart from timing and space consumption. To ensure this transparency, we require assertions to be *pure*: the execution of an assertion must not have any side-effects on the execution of the program.

2.2.4 Further reading

While the previous introduction to assertions gives the reader a sufficient understanding of the concepts that are used in this thesis, the interested reader is encouraged to learn more on the subject in the work of Clarke et al. [15] on the history of runtime assertion checking.

2.3 Multithreaded programs

Programs consist of *threads*: independent code executions that are running in parallel. A program with one thread is called *single-threaded*, while a program with multiple threads is *multithreaded*.

The execution of a single-threaded program is deterministic: the result of the program depends only on the external input of the program. The state of a multithreaded program is not only affected by its input, but also by the timing of the program: the statements of thread A can be executed before the statements of thread B during one execution of the program, while the statements of threads A and B alternate, or even execute simultaneously, in another execution.

Threads can share the resources of a program, such as files on the file system or objects that are created in the same program. Threads in Java have shared access to the *heap*: a section of the memory that contains all objects that are created by the program.

2.3.1 Interference

Concurrent modification of the same resource can lead to unpredictable behaviour in a program. Consider thread A changing a field of an object, and thread B making a change immediately afterwards. Thread A might expect its own value in the field, while it is actually the value written by B, which might lead to a failure in the program. Such unwanted interaction between program threads is called *interference*.

We can assign a direction to the interference of two threads. Interference will always be caused by a thread (e.g. B) writing to the heap, while the results of an unexpected change of the heap will be seen by a thread (e.g. A) reading this value. A thread will be *subject to interference* if the heap modification of another thread causes unexpected behaviour to the thread. A thread is *causing interference* if its actions result in another thread behaving unexpectedly.

An operation in a multithreaded program is *atomic* if the rest of the program can consider the execution of the operation as if it happens instantaneously. Interference can occur before or after, but not during the execution of an atomic operation. Most statements and expressions in Java are not executed atomically. The programmer should therefore be aware that a thread accessing the heap can be subject to interference and a thread modifying the heap can cause interference with another thread of the program.

2.3.2 Assertion interference

Assertions are often written in the form of expressions that are executed by the program. These expressions can be subject to interference if they are executed in a multithreaded program: heap modification in thread B can occur during the execution of an assertion in thread A, possibly causing a false assertion violation. The execution of an assertion should never cause interference: the evaluation of assertions should be transparent, and should therefore never modify parts of the heap that are visible to the rest of the program.

We define *program interference* as the interference in a multithreaded program that causes unexpected behaviour in a thread that is currently executing normal program statements. *Assertion interference* is defined as the interference during the evaluation of an assertion, that causes an assertion to behave unexpectedly.

2.4 Problem statement

Assertion interference can cause a runtime assertion checker to report an assertion violation in a program, or miss an assertion that could have been detected. We will show in Chapter 4 that these situations can happen when the behaviour of the program is correct according to its specification. A runtime assertion checker that is subject to assertion interference may therefore no longer be trustworthy. This leads us to our research question:

Can we create an environment in which we can perform runtime assertion checking in multithreaded programs without the risk of assertion interference?

This research will focus on a solution in the Java programming language. We will search for a way to stop assertion interference. Chapters 3 and 4 will discuss the possibility to execute assertions as if they happen atomically. Chapters 5 and 6 will use the results of these chapters to modify OpenJML, an existing runtime assertion checker for Java.

2.5 Related work

A lot of research on multithreaded runtime assertion checking is focused on the prevention of both program interference and assertion interference.

Rodríguez et al. [27] define a syntax to specify locks and other non-interference properties in the Java Modeling Language. Aurujo et al. [9] discuss the use of safepoints in a program: specific points where the program itself must guarantee protection against assertion interference. A program can acquire the locks that it needs to ensure this protection before it enters this safe point.

Separation logic [25] and fractional permissions [13] specify tokens that a thread must hold before it accesses a field of an object. These tokens allow only a single process at the time to write to the field, but allow multiple processes to read simultaneously. While the concept of separation logic are applicable to runtime assertion checking, this technique is currently primarily used in static checking.

Chapter 3

The STROBE framework

Assertion interference occurs when threads in a multithreaded program concurrently modify the shared heap. An obvious solution for this problem would be to temporarily stop all threads of the program during the evaluation of an assertion, except for the thread that executes the assertion itself. If all threads wait for the assertion to finish, they can not modify the shared heap and thus cannot change the outcome of the assertion. Such a solution requires an assertion to communicate with all other threads and request them to pause their work. This will change the timing of the program considerably. Furthermore, threads will finish their current operation before pausing, possibly preventing program interference that would normally occur in the program.

An alternative would be to hook the runtime assertion checker into the Java virtual machine and use this environment to suspend the threads of the program. Java has deprecated this thread functionality and marks the suspension and stopping of threads unsafe [4]:

> "Thread.suspend is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes."

Deadlocks certainly change the behaviour of the program, breaking the transparency of the runtime assertion checker.

Instead of suspending the threads, we can try to prevent the program from making modifications to the shared heap of the program while it is being used by the assertion. We could supply a copy of the entire heap to the assertion and perform our checks on this copy, while the other threads work on the "live" version of the heap.

The challenge in this solution is making the actual copy of the heap. This has to be done atomically: it is impossible to guarantee safety from assertion interference if a thread modifies the heap during the copying process. Making such a copy is a costly process: almost the entire state of a Java program is saved on the heap. Making a copy would double the memory consumption of the program. Heap writes from other threads will temporarily lock while this copy is being made, potentially freezing the program for some time.

There has been more research into the prevention of changes on the heap. Particularly, the research on the STROBE framework [7] shows many similarities with our research problem. This research proposes a strategy to speed up the time spent on assertion checking, by executing assertions in parallel of the running program. The framework schedules the assertions in the program to be executed on *checker threads*: threads that are designed to mimic the heap to

reflect a certain state of the program. The concurrently executed assertions in this framework are called *asynchronous assertions*.

The regular *program threads* in the STROBE framework will continue their execution while asynchronous assertions are executed on the checker threads. This will potentially expose asynchronous assertions to assertion interference from the program threads. The STROBE framework does not make a full copy of the heap to prevent this. Instead, it uses a *copy-on-write* strategy to preserve a certain state of the program: the framework only makes a copy of objects for the assertion if a program thread modifies the objects on the heap. These copies are saved in a *snapshot*: a special part of the heap, designed to keep track of the copies of an object that are needed to preserve a certain state of the heap.

The STROBE framework seems to provide a good solution for the prevention of assertion interference. We need to have a good understanding of the framework to evaluate if the framework can be used to modify existing runtime assertion checkers for the safe evaluation of assertions in multithreaded programs. This chapter will give a description of the STROBE framework and the underlying principles that ensure the correct evaluation of asynchronous assertions. We will discuss the possibilities and the limitations of the current framework, in order to determine what work needs to be done to use the framework for the interference free evaluation of assertions in multithreaded programs.

3.1 Introducing the STROBE framework

The STROBE framework provides a platform, designed on top of JikesRVM [8], to execute the assertions of a program on separate threads. JikesRVM is a Java Virtual Machine (JVM) that is almost completely written in Java. JikesRVM is designed for the research community, to aid in the experimentation with new language designs and implementation techniques. The virtual machine is extensible and is, among other things, used for the development of new garbage collection policies [11] and runtime optimization strategies [24]. STROBE uses the extensibility of JikesRVM to implement snapshots of the heap.

Asynchronous assertions in STROBE are wrapped in *futures* to schedule them on a separate thread. The framework uses snapshots to ensure a consistent state for these futures during the parallel execution of the assertions.

3.1.1 Futures

When a program starts the evaluation of an asynchronous assertion, the STROBE framework immediately returns a handle (often called a *future*). This future will eventually hold the result of the assertion. The assertion itself is scheduled for parallel execution. The program can use the future to request the result of its assertion execution. The future will return this result if the evaluation of the assertion is finished, and block until it has if this is not the case. In this way, a program can continue its normal execution as long as the results are not needed. We define the execution of the program between the start of the future and the joining point between the program thread and the future as the *continuation* of the future.

The first proposals for futures are for MultiLisp [20]. These futures were meant to speed up sequential code without any side effects to the program itself: the programmer should be able to use a future to execute a block of code as if it was executed sequentially. The Java language supplies futures in the `java.util.concurrent` package [1]. These futures enable parallel execution of code, but do not guarantee the interference free execution of either the future or its continuation. We would like to use the safety property that MultiLisp provided for asynchronous assertions

in Java: the observed result of a sequential assertion A_s is equal to the observed result of an asynchronous assertion A_a that is scheduled in its place.

Welc et al. [30] formalized a safety property for futures in Java, that allows for the execution of futures in place of their sequential counterpart, based on the Bernstein conditions [10]:

> "(1) an access to a location l (either a read or a write) performed by a future should not witness a write to l performed earlier by its continuation, and (2) a write operation to some location l performed by a future should be visible to the first access (either a read or a write) made to l by its continuation."

They propose a solution to maintain this safety property by keeping track of changes in the memory. A future is executed on a separate process. It can be restarted if it reads a location that is later modified by the continuation. The continuation can similarly be re-evaluated if a future modifies the shared state.

The pureness of assertions eases the safety properties of the futures that are needed to check asynchronous assertions. Futures should still provide safety property (1), but property (2) is automatically ensured by the fact that assertions will never change the state of the program. We therefore never need to restart the evaluation of the continuation: assertions will not modify any part of the state that is visible to the continuation.

3.1.2 Snapshots

The STROBE framework introduces a versioning system to objects on the heap. Every mutation of an object is seen as a new "version" of that object. The STROBE framework keeps track of the multiple versions of an object.

Asynchronous assertions are implemented using this versioning system. The STROBE framework keeps track of a global version number for the entire heap. It will save a snapshot of the program when an asynchronous assertion is scheduled. This will save the current version number of the heap.

When a snapshot is active, the STROBE framework saves the old version of an object as soon as the program tries to modify it (copy on write). Asynchronous assertions will use the version of this object that was present on the heap when the future was scheduled.

An asynchronous assertion uses snapshots to witnesses the state of the program at the time it was scheduled. We will call this state the *snapshot projection* of an asynchronous assertion. Every object access will be transparently rerouted to an access to the snapshot version of the object that belongs to the asynchronous assertion. In this way, asynchronous assertions satisfy part (1) of the safety property described in the previous section.

3.2 Using the STROBE framework

We will demonstrate the asynchronous assertions using the `Node` class of a linked list with an `addNode` method, as shown in Figure 3.1. The `addNode` method inserts the given node as a new successor of the current node and attaches the current successor to the given node. The list in this example is not allowed to contain the same node twice: this will create a cycle.

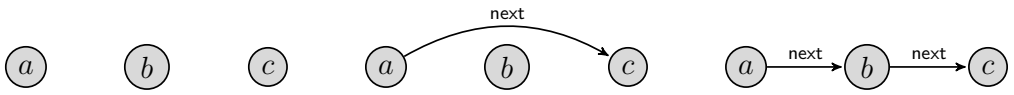
Figure 3.1 shows this method with standard assertions: the assert statement at line 5 acts as the precondition of this method and checks that the given node is not yet in the list. The statement at line 10 verifies that the execution of the method resulted in the node being part of the list. The `contains` method is used to check if a node is already in the list. The assertions

```
1  public class Node {
2    private Node next;
3
4    public void addNode(Node node) {
5      assert !this.contains(node);
6
7      node.next = this.next;
8      this.next = node;
9
10     assert this.contains(node);
11   }
12
13   @ConcurrentCheck
14   public boolean contains(Node node){
15     if(this.next == null)
16       return false;
17
18     else
19       return this.next == node
20         || this.next.contains(node);
21   }
22
23   public static Node createTestList(){
24     Node a = new Node();
25     Node b = new Node();
26     Node c = new Node();
27
28     a.addNode(c);
29     a.addNode(b);
30
31     return a;
32   }
33 }
```

Figure 3.1: addNode example with asynchrnous assertions

```
1  public void addNode(Node node) {
2    CHAFuture precontains =
3      new CHAFuture(new ContainsAssert(node));
4    precontains.go();
5
6    node.next = this.next;
7    this.next = node;
8
9    CHAFuture postcontains =
10     new CHAFuture(new ContainsAssert(node));
11   postcontains.go();
12
13   assert !precontains.get();
14   assert postcontains.get();
15 }
16
17 private class ContainsAssert
18   implements CHATask {
19
20   private Node appendNode;
21
22   public ContainsAssert(Node node){
23     appendNode = node;
24   }
25
26   @ConcurrentCheck
27   public boolean compute() {
28     return contains(appendNode);
29   }
30 }
```

Figure 3.2: addNode example with asynchronous assertions



(a) Start state (b) a.addNode(c) (c) a.addNode(b)

Figure 3.3: Building a list using createTestList

in this method are expensive. The insertion itself executes in constant time, but the assertions recursively check every node in the list to check if the list contains the given node.

The createTestList method creates a list with 3 elements: a, b and c (in that order). The creation of this list is shown in Figure 3.3.

Figure 3.2 shows how we can check the preconditions and postconditions of the *addNode* method with the use of asynchronous assertions. We replace the assert statement at line 5 of Figure 3.1 with a future on line 2-4. This future will notify a checker thread of the asynchronous assertion that needs to be checked and schedules it for parallel execution. The asynchronous assertion itself is defined in the compute method of the CHATask class, at line 27 of Figure 3.2. The future immediately returns a handle and an asynchronous checker thread is used to calculate the requested result while the method continues its execution. Finally, the handle is used on line 13 to query the future for the result of the asynchronous assertion.

Figure 3.4 shows the states of the list during the execution of the createTestList method, with asynchronous assertions enabled. States 3.4a, 3.4c and 3.4e correspond to the states of the memory, as observed in the method body of createTestList. These states are equal to the states observed during the execution of createTestList with synchronous assertions.

States 3.4b and 3.4d correspond to the state of the memory after the execution of line 7 of addNode in Figure 3.2. At this point, the addNode method has added the given node as its successor, while a snapshot of the previous version of the object is preserved for the asynchronous assertion. In state 3.4b, the asynchronous assertion (Figure 3.2, lines 2-4) will observe nodes (a_0, b_0, c_0) as the state of the program, while the program thread itself will observe the states (a_1, b_0, c_0). In this way, the asynchronous assertion observes that the program is in state 3.4a, while the program itself is operating on state 3.4c. The snapshots are marked for garbage collection when the execution of the asynchronous assertion is completed. State 3.4d is similar to state 3.4b: the assertion uses state 3.4c, while the program itself operates on 3.4e.

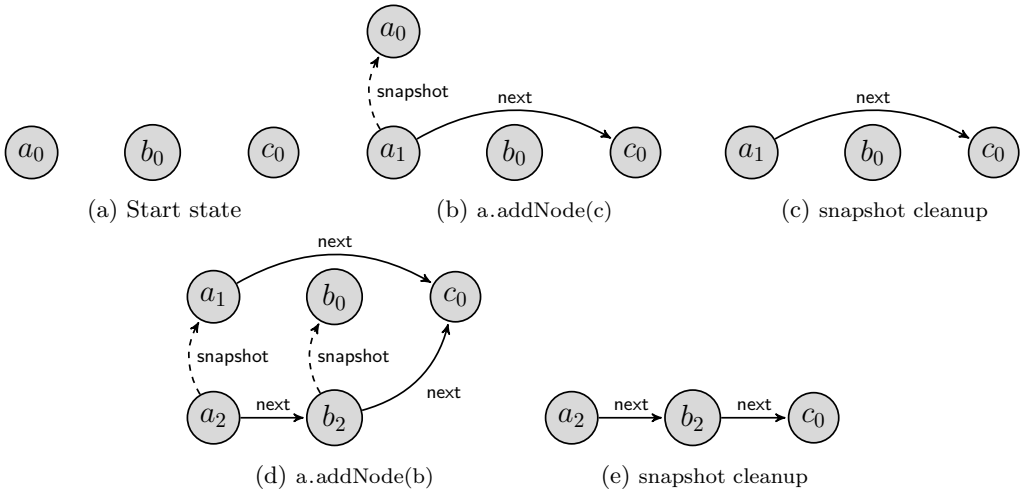
3.3 Semantics of snapshots

The introduction of snapshots changes the heap layout used by a program. This section will provide the semantics needed to reason about a heap with snapshots.

3.3.1 Single snapshot

The goal of a snapshot is to preserve the state of the heap as it was when the snapshot was taken. We first consider a program with a single snapshot. We look at the heap after a snapshot was taken, but before any modifications have taken place. The live state of the program will be equal to the snapshot state of the heap as long as this is the case. Therefore, no object needs to be copied.

When the program modifies an object to the heap, the live state after this modification will no longer be the same as the snapshot. Therefore, if the object is not yet preserved in the snapshot, a copy of the object will be made for this purpose.



(a) Start state (b) a.addNode(c) (c) snapshot cleanup

(d) a.addNode(b) (e) snapshot cleanup

Figure 3.4: Building a list using createTestList

When an asynchronous assertion attempts to read the value of an object field in a snapshot, the STROBE framework will check if a copy of the object has been made for this snapshot. The STROBE framework will return this copy if it exists. Otherwise, the live state is not modified and is used.

3.3.2 Multiple snapshots

A program can have multiple asynchronous assertions, which can run simultaneously. In this case, the STROBE framework maintains a snapshot for each asynchronous assertion. Consider a program with an active snapshot (snapshot 1), when another snapshot (snapshot 2) is taken.

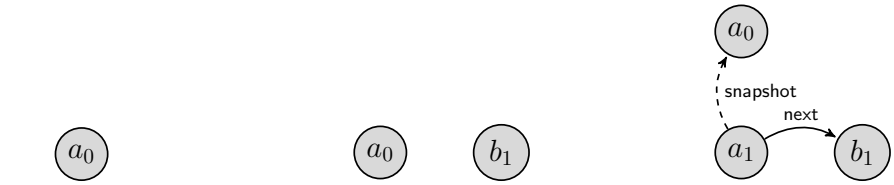
At this moment, the STROBE framework will contain a copy of all objects that were modified after snapshot 1 was taken, associated with snapshot 1. When the program tries to modify an object to the heap, the STROBE framework will now check both snapshots to see if the snapshot already contains a copy of this object, and otherwise save a copy.

3.3.3 Global versioning

The heap modification strategy described above will quickly become expensive when many snapshots are active: the framework needs to check for every snapshot if it either needs or already has a copy of an object. A global epoch counter E is used to solve this problem. The counter will be initialized to 0 when the program is started and increments for every snapshot that is taken.

For every object on the heap, the STROBE framework saves the epoch in which the object is last modified. Object access in a snapshot can now check this epoch to see if it needs to access the live version of an object, or its snapshot: if the epoch of the live snapshot is equal to the epoch of the snapshot, the live version can safely be used.

The epoch is also used when the program tries to modify an object on the heap. A snapshot only needs a copy of the object if the object was last modified before the snapshot was taken. Therefore a snapshot might need a copy of an object if the epoch of the object is smaller than the global epoch of the program. Similarly, copying can be skipped if the epoch of an object is



(a) Start state (b) Node b = **new** Node() (c) a.addNode(b)

Figure 3.5: Building a list using createTestList

equal to the current epoch of the program. Figure 3.4 shows the epoch counters for the nodes in the linked list.

3.3.4 Guarantees

The snapshot strategy described above guarantees that a snapshot will always see the heap in the state that was present when the snapshot was made, while objects are only copied when necessary. Object copies are made atomically: when an object is read by a snapshot, it cannot be written, and when an object is written, an asynchronous assertion has to wait before the object can be read.

Snapshots will have the same object reference as the live version of the object. This means that the program is unaware of the use of snapshots: it treats every object in a snapshot as if it is a normal object. As a result, the references to an object do not need to be changed when a snapshot is taken for the object: the framework transparently reroutes object access to the snapshot.

3.3.5 Creating new objects

A program thread can create new objects after a snapshot has been taken. These new objects can not be reached by the snapshot: they were not part of the heap when the snapshot was taken. This reachability property can be verified by looking at the reachability of a newly created object. Consider the following program, that uses the linked list of Figure 3.1:

```
Node a = new Node();
CHAFuture contains = new CHAFuture(new ContainsAssert(node));
contains.go();
Node b = new Node();
a.addNode(b);
```

Figure 3.5 shows the different states of this example. The object b on the heap can only be reached if another object refers to it. As soon as the program tries to store a reference to this newly created object in an object a, a copy a_0 will be made of the object a. The checker threads work with this snapshot copy, that does not contain the reference between a and the newly created object b. Therefore, b can not be reached in the snapshot.

The reachability property of newly created objects in a snapshot allows for an optimization: the STROBE framework does not need to keep a snapshot copy of newly created objects in the forwarding array. The epoch counter provides this optimization automatically: snapshots only copy objects that were modified before the snapshot was taken, and the epoch counter of the newly created object is the current epoch, so none of the current snapshots will maintain a copy for this object.

The heap separation works both ways. An asynchronous assertion can create new objects, to save its intermediate results. These objects are completely separated from the reachable heap of the program thread, as the live heap does not contain any reference to this newly created object.

3.4 Implementation

3.4.1 Barriers

The STROBE framework monitors for write actions when a snapshot is active. It uses *object barriers* in JikesRVM to do so. These barriers are special blocks of code that are used to perform extra work when the virtual machine reads from or writes to the heap [12]. Barriers can either be *substituting* or *non-substituting*. Substituting barriers are compiled in place of the heap action, and are therefore responsible for the correct execution of the heap operation. Non-substituting barriers are compiled before and/or after the heap operations, wrapping the operation with barrier code, but not replacing the operation itself. Write barriers are always substituting, while read barriers can be either substituting or non-substituting. STROBE only uses substituting barriers.

The STROBE framework uses write barriers to create a copy of an object, if a snapshot needs such a copy to maintain its state. This can result in multiple copies of an object: the STROBE framework creates a copy for every available checker thread, if necessary. Copies are shared between snapshots if the snapshot state is the same for that object.

A method annotated with the `@ConcurrentCheck` annotation will be compiled with substituting read barriers. Object access in such a method is preceded by a snapshot lookup. If the current thread is operating on a snapshot, the snapshot copy of the object is returned instead of the object itself.

The STROBE framework will only compile the read barriers into methods that are annotated with a `@ConcurrentCheck` annotation. This means that the `contains` call on line 28 of Figure 3.2 will actually fail to produce the correct result. This bug in our example is fixed by annotating the `contains` method on line 14 of Figure 3.1 with a `@ConcurrentCheck` annotation. The need to explicitly annotate every method used in assertions limits the usefulness of assertions: we can not use built-in Java classes, such as `java.util.ArrayList`, as these classes can not be annotated with the `@ConcurrentCheck` annotation.

3.4.2 Object headers

Every object in the JikesRVM consists of an object header and a set of fields. The object header contains meta information about the object, such as an identifier, type information and the lock state of the object. This object header can be extended with custom fields, that can be accessed from the virtual machine. The STROBE framework creates a custom field that is used to maintain a *forwarding array* of snapshot copies of the object. This forwarding array contains references to the snapshots that are made for this object. The write barriers update this forwarding array if snapshots are active. The read barriers in turn use the array to return a reference to a snapshot version instead of the object itself, as described in Section 3.3.

3.4.3 Checker threads

When a program thread schedules an asynchronous assertion, the work is delegated to a separate checker thread. The STROBE framework uses a dedicated pool of checker threads, allowing for multiple assertions to run in parallel. The maximum number of checker threads is limited, and

can be configured when the virtual machine is started. Each checker thread has its own *checker identifier*, saved in an instance of the CheckerThread class.

When an asynchronous assertion is scheduled, it searches for a free checker thread, or waits until a thread is finished with its work. The checker identifier of the selected checker thread is then used to create the snapshot for the assertion. The future that contains the asynchronous assertion is saved in the CheckerThread instance of the checker thread, and the thread is activated. Figure 3.6a shows the process of waking up a checker thread in pseudo-code.

The checker thread will start the execution of the assertion when it wakes up, by calling the compute() method of the asynchronous assertion. The STROBE framework will recognize the @ConcurrentCheck annotation of this method and executes this method against a snapshot projection of the heap. The result of the assertion (success or failure) is then saved in the associated CHAFuture, giving the program access to this result. Afterwards, the checker thread switches back to the live state of the heap, wakes up any program threads that are waiting for the result, and waits for the next asynchronous assertion to execute. Figure 3.6b shows the pseudo-code for this checking process.

3.5 Limitations

3.5.1 Native and reference types

The current implementation only enables barriers for object reference types. The framework does not place barriers around the access of native typed fields. While this is inconvenient, a program using native types can be easily modified to use object references, by explicitly using the boxed version (e.g. *Integer*) of a native type. This only needs to be changed at the declaration of a variable. Java will automatically box any native values that will be assigned to this variable.

3.5.2 Static fields

Every class in a Java program can contain *instance fields*, that are associated with an instance of the class, and *static fields* that are associated with the class itself. The amount of memory that is needed for these static fields is known at the start of the execution of program. Therefore, JikesRVM saves these fields in a special part of the memory, called the *JTOC register*. The current implementation of the STROBE framework does not enable snapshot barriers for static fields. Creating these static barriers would require a strategy that differs from copying objects: either the entire JTOC register needs to be copied for the snapshot, or each static field needs to have its own forwarding array to keep track of the copies.

3.5.3 Assertion interference in multithreaded programs

Figure 3.7b shows an asynchronous assertion that checks if two nodes are equal. This asynchronous assertion is used in the equalsTest method in Figure 3.7a to compare the result of getNext() with itself. This assertion in this method will always pass in a single-threaded program: the *getNext()* method returns the same result twice and the asynchronous assertion will report that these nodes are equal to each other.

In a multithreaded program, a second program thread may be able to access the Node, while the equalsTest method is executed. This second program thread can change the Node, to let the next instance variable point to a different Node. The assertion will fail if this happens between the two getNext() calls on line 11. The STROBE framework will not provide protection against this form of assertion interference: the snapshot that is used for the future is created after the

```
1  public void startMe(CHAFuture future)
2  {
3    // wait for a checker to become available;
4    CheckerThread checker =
5      waitForFreeChecker();
6
7    // assign asynchronous assertion
8    checker.checkToDo = future;
9
10   // initiate assertion
11   Snapshot.initiateProbe(checker.checkerId);
12
13   // wake up checker thread
14   checker.wakeUp();
15 }
```

(a) Waking up a checker thread

```
1  public void run(){
2    while (true) {
3      // wait for the next task
4      waitForAsynchronousAssertion();
5
6      // compute assertion
7      checkToDo.result = checkToDo.compute();
8
9      // clean up snapshot
10     Snapshot.completeProbe(checkerId);
11
12     // notify waiting program threads
13     checkToDo.notifyAll();
14
15     // prepare for the next assertion
16     checkToDo = null;
17   }
18 }
```

(b) Executing an asynchronous assertion

Figure 3.6: Executing an asynchronous assertion

```
1  public class Node {
2    private Node next;
3
4    private Node getNext(){
5      return next;
6    }
7
8    public void equalsTest(){
9      CHAFuture equalsFuture =
10       new CHAFuture(new EqualsAssert(
11         this.getNext(), this.getNext()
12       ));
13     equalsFuture.go();
14     assert equalsFuture.get();
15   }
16 }
```

(a) Skeleton of the Node class

```
1  private class EqualsAssert
2          implements CHATask {
3
4      private Node a;
5      private Node b;
6
7      public EqualsAssert(Node a, Node b){
8          this.a = a;
9          this.b = b;
10     }
11
12     @ConcurrentCheck
13     public boolean compute() {
14         return a == b;
15     }
16 }
```

(b) Asynchronous assertion

Figure 3.7: Initializing an asynchronous assertion task

```
1  public class Timer {
2
3    private Integer time = 0;
4
5    public int getTime(){ return time; }
6
7    public void increase(){
8      Integer oldTime = this.getTime();
9
10     time = time + 1;
11
12     assert this.getTime() > oldTime;
13   }
14 }
```

Figure 3.8: Assertions on state transitions

initialization of the CHATask that will execute the assertion. The framework only protects the execution of the asynchronous assertion, not its creation. The programmer is responsible for the prevention of assertion interference during the creation of futures.

3.5.4 State transitions

Assertions can check state transitions by comparing two different states of the program with each other. Typically, the program state before the execution of program code (pre-state) is compared to the program state after the execution. Runtime assertion checkers normally only have access to the current state of the program, and will therefore pre-calculate the parts of the pre-state that are required to execute such an assertion. The assertion itself is placed in the postcondition, and uses the pre-calculated information instead of the actual pre-state of the program.

The Timer class in Figure 3.8 gives an example of an assertion on a state transition. The timer has the property that its value will only increase during program execution. The increase method checks this property. The method pre-calculates the timer value in the pre-state at line 8. The assertion itself, on line 12, reads the current value of the timer and compares it with the pre-calculated value.

Asynchronous assertions are executed on a single state of the program. It is therefore not possible to protect both the pre-state and the post-state against assertion interference. Figure 3.9 shows an implementation of the Timer class with an asynchronous assertion. The pre-state is still pre-calculated, and passed to the asynchronous assertion at initialization. The assertion itself is evaluated after this initialization. This kind of pre-calculation is therefore still subject to assertion interference, as we have shown in the previous section.

3.6 Conclusion

The STROBE framework allows us to evaluate asynchronous assertions on a specific state of a program. The framework protects assertions against interference: the STROBE framework provides the statements of an assertion with a projection of the heap as it was when the assertion was scheduled.

Assertions in the STROBE framework are written in specialized objects, that are scheduled for execution on a different thread. The framework does not provide protection against assertion interference during the initialization of these objects.

```
1  public class Timer {
2    private Integer time = 0;
3
4    @ConcurrentCheck
5    public int getTime(){ return time; }
6
7    public void increase(){
8    CHAFuture increasingFuture =
9      new CHAFuture(new IncreasingAssert(this.getTime()));
10
11     time = time + 1;
12
13     increasingFuture.go();
14     assert increasingFuture.get();
15   }
16
17   private class IncreasingAssert implements CHATask {
18     private Integer oldTime;
19
20     public IncreasingAssert(Integer oldTime){
21       this.oldTime = oldTime;
22     }
23
24     @ConcurrentCheck
25     public boolean compute() {
26       return getTime() > oldTime;
27     }
28   }
29 }
```

Figure 3.9: Assertions on state transitions

In this research, we search for a solution to safely perform runtime assertion checking for multithreaded programs. Such a solution requires protection from assertion interference during the complete evaluation of assertions. In the case of asynchronous assertions, this means protection during both the creation and the execution of asynchronous assertions.

The current STROBE framework therefore does not provide a complete solution for our research problem, but can be modified to suit our needs. We will discuss our modifications in the next chapter.

Chapter 4

The Extended STROBE framework

The STROBE framework will prevent assertion interference during the execution of the assertion. The framework does not prevent assertion interference during the initialization: initialization of an asynchronous assertion is done before the snapshot for the assertion is taken. A multithreaded program is therefore not completely protected against assertion interference.

We can prevent assertion interference during the complete evaluation of assertions if the statements for both the initialization and the execution of the asynchronous assertion use the same snapshot projection of the heap for their execution.

The current STROBE framework encapsulates the creation of snapshots in the *CHAFuture.go()* method call, which is used to schedule the asynchronous assertion. This disables the possibility to switch to a snapshot projection of the heap during the initialization of the asynchronous assertions. Exposing the snapshot creation and modification to the programmer would allow us to switch to a snapshot earlier, and perform the initialization of an asynchronous assertion on the snapshot projection of the heap.

This chapter will discuss the *Extended STROBE framework* (e-STROBE): our extension to the STROBE framework that exposes the snapshots directly to the program. Programmers will get the ability to choose whether statements in a program are executed on the live state of the program or on a snapshot projection of the heap. We will show how this ability can be used to completely protect assertions against assertion interference.

The e-STROBE framework enables programs to protect both synchronous and asynchronous assertions from assertion interference. The focus in this research will be on the use of snapshots for the protection of synchronous assertions, but the e-STROBE framework will still support asynchronous assertions. This allows us to run the benchmarks of the STROBE framework against our e-STROBE framework, and allows programmers to choose between the simplicity of synchronous assertions and the performance gains of asynchronous assertions.

Build and usage instructions for the e-STROBE framework can be found in Appendix A. This appendix also contains the location to the source code and binaries for this framework.

4.1 Exposing snapshots to the program

The STROBE framework executes the asynchronous assertions against a snapshot projection of the heap. The framework uses the checker identifier to create a snapshot for the checker thread that is going to execute the asynchronous assertion. The snapshot is directly coupled with the checker thread that uses it, only allowing snapshots to be used by checker threads.

The e-STROBE framework decouples the snapshots from checker threads, allowing program threads to create and use snapshots in their own code. The framework uses *snapshot identifiers* to keep track of the snapshots that are active in the program. The framework will associate a free snapshot identifier with the snapshot upon creation, and return this identifier to the caller. A snapshot can be initialized using the following statement:

```
snapshotId = Snapshot.initiateProbe();
```

The checker threads in the STROBE framework use the checker identifier of their CheckerThread instance, in conjunction with the @ConcurrentCheck annotation of the compute method, to decide that heap access should reflect the snapshot projection of the heap. This strategy limits the use of snapshots to checker threads.

The e-STROBE framework has a snapshot identifier field in the RVMThread class: a class in JikesRVM that is used to save extra information for the thread of the program. Every thread has an RVMThread associated with it. A thread in a program can gain access to its RVMThread instance using the RVMThread.getCurrentThread() method. The snapshot identifier in the RVMThread is public. Changing this identifier allows a thread to switch to the execution of statements on a snapshot projection of the heap:

```
RVMThread currentThread = RVMThread.getCurrentThread();
currentThread.snapshotId = snapshotId;
```

Changing the snapshot identifier to the value -1 changes the thread back to normal execution, against the live state of the heap:

```
currentThread.snapshotId = -1;
```

Programs in the e-STROBE framework have the responsibility to destroy their own snapshots. Destroying a snapshot will trigger the cleanup of all snapshot copies that were made for the snapshot projection of the heap, and recycles the snapshot identifier for future use. A program can destroy a snapshot using the following statement:

```
Snapshot.completeProbe(snapshotId);
```

Figure 4.1 shows how the snapshot interface described above can be used to protect the preconditions and postconditions of the addNode method in Figure 3.1 against assertion interference. The assertions at point A and C of the method will both be executed against a snapshot of the program, taken at respectively line 4 and line 16.

The example shows that a thread can create multiple snapshots, but it can use only one snapshot projection at the time. The explicit switching between snapshots allows us to identify regions of the program that are executed against a snapshot projection of the heap. We define statements in these regions within the *scope* of the associated snapshot. Statements outside a snapshot scope are in the *live scope* of the program. Figure 4.1 shows statements that are executed in 3 different scopes: assertions A and C executed in the scope of their respective snapshot and the normal method body B executed in the live scope of the program.

The snapshots scopes in this research are used to encapsulate assertions, and ensure that they are executed free of assertion interference. Therefore, we will always start and end a snapshot scope in the same method. The e-STROBE framework does not restrict snapshot scopes to this use: a program can switch to a snapshot in method A, call method B, and switch back to the live scope in method B. We discourage such use of snapshot scopes, as this practice is not necessary for the protection of assertions and makes programs harder to understand.

```
1   @ConcurrentCheck
2   public void addNode(Node node) {
3
4     int preconditionId = Snapshot.initiateProbe();
5     RVMThread currentThread = RVMThread.getCurrentThread();
6     currentThread.snapshotId = preconditionId;
7
8     assert !this.contains(node); // A
9
10    currentThread.snapshotId = -1;
11    Snapshot.completeProbe(preconditionId);
12
13    node.next = this.next; // B
14    this.next = node;      // B
15
16    int postconditionId = Snapshot.initiateProbe();
17    currentThread.snapshotId = postconditionId;
18
19    assert this.contains(node); // C
20
21    currentThread.snapshotId = -1;
22    Snapshot.completeProbe(postconditionId);
23  }
```

Figure 4.1: Assertion checking with the e-STROBE framework

4.2 Using snapshots in multithreaded programs

We have stated in Section 2.4 that assertion interference can cause a runtime assertion checker to report an assertion violation in a program while the behaviour of the program is correct according to its specification. We will give an example of a correct program that triggers such behaviour using a Timer and Display class, shown in Figure 4.2. The Timer class simulates a timer, counting from 1 to 10,000. The display class uses this timer to return the current value of the timer to the user. A single timer is tied to two displays in the testDisplay method. The timer is started in a separate thread. The testDisplay method will then assert that both displays will always display the same value.

In a single-threaded program, the assertion on line 35 of Figure 4.2 will always be true: both displays return the time from the same timer. This result can change in a multithreaded program, as the assertion is not executed atomically. The timer can increment its value between the method calls d1.getTime() and d2.getTime(), leading to the report of an assertion violation by the runtime assertion checker. This is a false report, caused by assertion interference: the two displays will reflect the same timer state in every state of the program.

Figure 4.3 shows the use of snapshots in the testDisplay method of the Display class to eliminate this interference. The method will now execute the assertion on line 15 in a snapshot scope, causing both the d1.getTime() and d2.getTime() method calls to be executed on the same state of the Timer. Therefore, the incrementing value of the timer in the live state of the program can no longer cause an assertion violation.

Assertion interference can also mask an assertion violation in a program. We create such a situation using a broken display, shown in Figure 4.4. This broken display will always return its time, minus one. Therefore, the two displays will never show the same value. However, the assertion on line 15 might pass when the value of the timer gets incremented between the execution of d1.getTime() and d2.getTime(). Again, placing the assertion in a snapshot scope will remove this assertion interference. This will correctly trigger an assertion violation in every execution of the program.q

```
1  public class Timer extends Thread {
2
3    private Integer time = 0;
4
5    @ConcurrentCheck
6    public int getTime(){ return time; }
7
8    @Override
9    public void run(){
10     for(int i = 0; i < 10000; i++){
11       this.time = this.time + 1;
12     }
13   }
14 }
15
16 public class Display {
17   private Timer timer;
18
19   public Display(Timer timer){
20     this.timer = timer;
21   }
22
23   @ConcurrentCheck
24   public int getTime(){
25     return timer.getTime();
26   }
27
28   public static void testDisplay(){
29     Timer t = new Timer(barrier);
30     t.start();
31
32     Display d1 = new Display(t);
33     Display d2 = new Display(t);
34
35     assert d1.getTime() == d2.getTime();
36   }
37 }
```

Figure 4.2: Timer and Display class

```
1
2     @ConcurrentCheck
3     public static void testDisplay(){
4       Timer t = new Timer();
5       t.start();
6
7       Display d1 = new Display(t);
8       Display d2 = new Display(t);
9
10      RVMThread currentThread = RVMThread.getCurrentThread();
11
12      int snapshotId = Snapshot.initiateProbe();
13      currentThread.snapshotId = snapshotId;
14
15      assert d1.getTime() == d2.getTime();
16
17      currentThread.snapshotId = -1;
18      Snapshot.completeProbe(snapshotId);
19    }
```

Figure 4.3: Using snapshots for assertions

```
1  public class BrokenDisplay extends Display{
2    public BrokenDisplay(Timer timer){ super(timer); }
3
4    //@ConcurrentCheck
5    public int getTime(){
6      return super.getTime() - 1;
7    }
8
9    public static void testDisplay(){
10     Timer t = new Timer();
11     t.start();
12
13     Display d1 = new Display(t);
14     Display d2 = new BrokenDisplay(t);
15
16     assert d1.getTime() == d2.getTime();
17   }
18 }
```

Figure 4.4: Creating a false negative with a broken display

```
1  public class Timer extends Thread {
2    private Integer time = 0;
3
4    @ConcurrentCheck
5    public int getTime(){ return time; }
6
7    @ConcurrentCheck
8    public void increase(){
9      RVMThread currentThread = RVMThread.getCurrentThread();
10
11     int preId = Snapshot.initiateProbe();
12     currentThread.snapshotId = preId;
13
14     Integer oldTime = this.getTime(); // A
15
16     currentThread.snapshotId = -1;
17     Snapshot.completeProbe(preId);
18
19     time = time + 1;
20
21     int postId = Snapshot.initiateProbe();
22     currentThread.snapshotId = postId;
23
24     assert this.getTime() > oldTime; // B
25
26     currentThread.snapshotId = -1;
27     Snapshot.completeProbe(postId);
28   }
29 }
```

Figure 4.5: Pre-calculation before execution

4.3 State transitions

4.3.1 Pre-state calculation

We will revisit the Timer example that was used in Figure 3.8. The STROBE framework could protect the assertion in this example from assertion interference with the use of asynchronous assertions, but we were unable to provide the same protection to the pre-calculation of the timer value.

The e-STROBE framework allows for the use of multiple snapshot scopes in the same method. This allows us to use the local variables of a method to transfer intermediate results between snapshots. The e-STROBE framework will not create snapshot copies for these local variables: they can only be used by the method itself, and will not be shared between threads.

Figure 4.5 shows an implementation of the Timer with snapshots to protect the state transition assertion. The value of the timer is calculated in the snapshot scope at A and saved in the local oldTimer variable. The assertion itself is executed in the snapshot scope at B, after the timer has been incremented. The value of the oldTimer variable is still available in this scope and is compared to the value of the post-state copy of the timer.

We do not have to calculate the pre-state immediately after we take the snapshot: snapshots remain valid until they are destroyed by the program. This fact is exploited in Figure 4.6 to create an alternative implementation of the increase() method of the Timer. This version increments the timer first, and saves the oldTimer value afterwards. The snapshot scope ensures that a snapshot copy of the timer from the pre-state is used in the calculation of the oldTimer value, and the

```
1  public class Timer extends Thread {
2    private Integer time = 0;
3
4    @ConcurrentCheck
5    public int getTime(){ return time; }
6
7    @ConcurrentCheck
8    public void increase(){
9      RVMThread currentThread = RVMThread.getCurrentThread();
10
11     int preId = Snapshot.initiateProbe();
12
13     time = time + 1;
14
15     int postId = Snapshot.initiateProbe();
16
17     currentThread.snapshotId = preId;
18     Integer oldTimer = this.getTime(); // A
19
20     currentThread.snapshotId = postId;
21     assert this.getTime() > oldTimer.getTime(); // B
22
23     currentThread.snapshotId = -1;
24     Snapshot.completeProbe(preId);
25     Snapshot.completeProbe(postId);
26   }
27 }
```

Figure 4.6: Pre-calculation after execution

assertion will therefore still be evaluated correctly.

It is important to pay attention to the types of local variables that are used to transfer state between snapshot scopes. We can safely use native typed local variables: their values will be directly associated with the variable. However, we have to be careful if a local variable contains an object reference.

Figure 4.7 shows an implementation of the increase method that does not pre-calculate the value of the timer at point A. Instead, the oldTimer variable contains a reference to the Timer object itself. This reference is used in the snapshot scope at B, but the timer that is referenced is an object on the heap. In fact, **this** and oldTimer will point to the same timer object on the heap, and the evaluation of this assertion will lead to an assertion violation. In this case, the program could have made a copy of the timer in the pre-state, manually saving the state of Timer object for use in the post-state.

4.3.2 Evaluating the result of a method

Snapshots allow us to evaluate of statements against a previous state. This allows us to write a new type of assertion, that we could previously not evaluate with a runtime assertion checker.

Figure 4.8 extends the Node class in Figure 3.1 with a removeNode method and a getNode helper method. The removeNode method removes the given node from the list and returns its index, observed from the node on which the method was called. This method contains two assertions, similar to the assertions in addNode: the assertion on line 4 of Figure 4.8, checking if the node is present in the list and the assertion on line 15, checking if the node was deleted from the list. Note that node removal is subject to program interference. We will only consider

```
1  public class Timer extends Thread {
2    private Integer time = 0;
3
4    @ConcurrentCheck
5    public int getTime(){ return time; }
6
7    @ConcurrentCheck
8    public void increase(){
9      RVMThread currentThread = RVMThread.getCurrentThread();
10     int preId = Snapshot.initiateProbe();
11     time = time + 1;
12
13     int postId = Snapshot.initiateProbe();
14     currentThread.snapshotId = preId;
15     Timer oldTimer = this; // A
16
17     currentThread.snapshotId = postId;
18     assert this.getTime() > oldTimer.getTime; // B
19
20     currentThread.snapshotId = -1;
21     Snapshot.completeProbe(preId);
22     Snapshot.completeProbe(postId);
23   }
24 }
```

Figure 4.7: Incorrect use of object references in state transitions

```
1  // Removes a node from the list
2  // and returns the index
3  public int removeNode(Node node) {
4    assert this.contains(node);
5
6    int index = 1;
7    Node nodeToCheck = this;
8    while(nodeToCheck.next != node){
9      nodeToCheck = nodeToCheck.next;
10     index = index + 1;
11   }
12   nodeToCheck.next = nodeToCheck.next.next;
13   node.next = null;
14
15   assert !this.contains(node);
16
17   return index;
18 }
19
20 // returns the nth entry in the list
21 @ConcurrentCheck
22 public Node getNode(int index){
23   if(index == 1)
24     return this.next;
25   else
26     return this.next.getNode(index - 1);
27 }
```

Figure 4.8: Node removal

```
1  // Removes a node from the list
2  // and returns the index
3  @ConcurrentCheck
4  public int removeNode(Node node) {
5    assert this.contains(node);
6
7    int snapshotId = Snapshot.initiateProbe();
8
9    int index = 1;
10   Node nodeToCheck = this;
11   while(nodeToCheck.next != node){
12     nodeToCheck = nodeToCheck.next;
13     index = index + 1;
14   }
15   nodeToCheck.next = nodeToCheck.next.next;
16   node.next = null;
17
18   assert !this.contains(node);
19
20   RVMThread currentThread =
21     RVMThread.getCurrentThread();
22   currentThread.snapshotId = snapshotId;
23
24   assert this.getNode(index) == node;
25
26   currentThread.snapshotId = -1;
27   Snapshot.completeProbe(snapshotId);
28
29   return index;
30 }
```

Figure 4.9: Node removal with snapshots

```
1       RVMThread currentThread = RVMThread.getCurrentThread();
2
3       int snapshotA = Snapshot.initiateProbe();
4       currentThread.snapshotId = snapshotA;
5
6       int snapshotB = Snapshot.initiateProbe();
7
8       currentThread.snapshotId = -1;
9
10      Snapshot.completeProbe(snapshotA);
11      Snapshot.completeProbe(snapshotB);
```

Figure 4.10: Nested initialization of snapshots

node removal for single-threaded programs, as this is sufficient to demonstrate this new type of assertion.

We want to write an assertion that checks if the value that we return at the end of this method pointed to the index of the removed node. We should examine this index in the preconditions of the method, as the execution of the method will ensure that the node is no longer in the list. The index, however, is only available after the execution of the method.

Figure 4.9 shows a version of the removeNode method that uses a snapshot to delay the evaluation of the assertion until the index that we need is available. The program takes a snapshot at line 7, and then performs the deletion. The desired assertion on the index of the executed Node is performed on line 24, inside the snapshot scope of the pre-state. The index of the deleted object is available at this point, and is used by the assertion.

4.4 Extended semantics of snapshots

The snapshots in the e-STROBE framework will inherit the semantics of the STROBE framework, described in Section 3.3. However, the snapshot interface introduced in Section 4.1 makes it possible to break the isolation of the execution of snapshots, that is present during the execution of asynchronous assertions. This section describes the extended semantics of snapshots with a description of our new snapshot interface, and describe the behaviour of a program that accidentally accesses object in snapshots scope that are not in the associated snapshot projection.

4.4.1 Initializing snapshots

Initializing a snapshot will trigger the e-STROBE framework to begin with the incremental preservation of the state of the heap at the moment of creation. The initialization of a snapshot is atomic, and happens transparently: the observed behaviour of the heap outside of snapshot scopes of this snapshot before the initialization of the snapshot is identical to the observed behaviour after the initialization.

Initializing a snapshot will always trigger the preservation of the live state of the heap. Initializing a snapshot within a snapshot scope, as shown in Figure 4.10, will result in a snapshot that includes the modifications that were made by all threads during the execution time that passed between the initialization of the snapshot on line 3 and 6.

4.4.2 Destroying snapshots

When a snapshot is destroyed, the program will stop the incremental preservation of its heap state. All objects copies that were preserved for the snapshot projection will no longer be associated with the snapshot. The e-STROBE framework will garbage collect these copies, unless they are still in use to preserve the heap state of another snapshot. The identifier of the snapshot is no longer valid after the destruction of the snapshot.

Just like initialization, the destruction of a snapshot is atomic, and happens transparently: the observed behaviour of the heap outside of snapshot scopes of this snapshot before the destruction of the snapshot is identical to the observed behaviour after the destruction.

4.4.3 Sharing snapshots between threads

Snapshots will preserve a snapshot projection of the heap as it was at the time of initialization. This projection is not bound to a certain thread: it is valid in all threads of the program. Snapshot identifiers can therefore be shared between the threads of the programs.

Asynchronous assertions are implemented using the sharing of snapshot identifiers. A snapshot is initialized in the program thread that requests the execution of the asynchronous assertion, and used by the checker thread to execute the asynchronous assertion in a snapshot scope.

4.4.4 Snapshot lifetime

A snapshot identifier is valid from the moment of initialization until the destruction of the snapshot. We define this period as the *lifetime* of a snapshot. Snapshot scopes of the associated snapshot are only valid within the lifetime of the snapshot. All snapshot scopes must therefore start after the creation of a snapshot, and finish before the snapshot is destroyed.

The e-STROBE framework does not check if a snapshot identifier is still in use when a snapshot is destroyed. The program is responsible for creating, using and destroying snapshots in the correct order.

4.4.5 Snapshot scopes

The e-STROBE framework uses a snapshot projection of the heap inside the snapshot scope. The framework guarantees that all accesses to objects that exist in this projection will be redirected to a version of the object that represents the state of the object at the time of snapshot initialization.

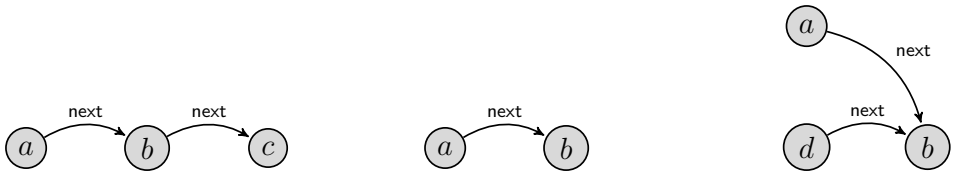
Objects that are created after the initialization of a snapshot are not included in the snapshot projection. Snapshot scopes will use the live state of these objects. These objects can therefore change during the evaluation of the snapshot scope, making assertions that use them subject to assertion interference.

A program thread executing on a snapshot scope is not allowed to change objects that exists in its snapshot projection, but the e-STROBE framework will not check this. Changing these objects will not change the version of the object that is in the snapshot projection, but will update the live state of an object, leading to unpredictable behaviour in the program. Making such changes is not necessary: assertions are transparent, and should therefore not make any changes that are visible to the rest of the program.

It is allowed to create new objects from within a snapshot scope. These new objects are created after the initialization of a snapshot, and are therefore not included in the snapshot projection. In this case, there is no risk of assertion interference: at the time of creation, the current thread is the only thread with a reference to the newly created object. The newly created

```
1       RVMThread currentThread = RVMThread.getCurrentThread();
2
3       Node a = new Node();
4       Node b = new Node();
5       Node c = new Node();
6       a.addNode(b);
7       b.addNode(c);
8
9       int snapshotId = Snapshot.initiateProbe();
10
11      b.removeNode(c);
12      d = new Node();
13      d.addNode(b);
14
15      currentThread.snapshotId = snapshotId;
16
17      assert d.next.next == a.next.next;
18
19      currentThread.snapshotId = -1;
20      Snapshot.completeProbe(snapshotId);
21
22  }
```

Figure 4.11: Assertion checking on an invalid program state



(a) Initial list (b) b.removeNode(c) (c) d.addNode(b)

Figure 4.12: List modification

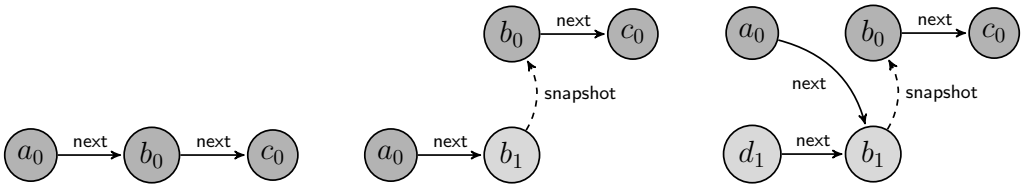
object can therefore only be accessed by the current thread, ruling out any possibility of assertion interference.

4.4.6 Using local variables

Local method variables can give a program executing a snapshot scope access to new objects that are created outside of the snapshot scope. These objects are also not included in the snapshot projection. Using these local variables enables a program to execute statements that use objects from both the snapshot projection and the live state of the program. In that case, the e-STROBE framework can no longer guarantee that such a statement is evaluated against a valid state of the program.

To illustrate this, Figure 4.11 shows a program that contains an assertion that is evaluated against an invalid state. The program uses the Node class described in Figure 4.1 and 3.1 to create a list with three Nodes: a and b and c. The programmer intended to modify the list by replacing Node a with Node d. He makes a mistake, and also removes c from the list. The correctness of this faulty replacement is checked by an assertion, that is executed in the scope of the previously taken snapshot.

Figure 4.12 shows the several states of the list during the execution of the program. Fig-



(a) Initial list (b) b.removeNode(c) (c) d.addNode(b)

Figure 4.13: List modification with snapshots

ure 4.13 shows the same states, including the snapshot copies that are made by the e-STROBE framework. The nodes that are included in the snapshot projection are highlighted. Nodes a and b and c are included in the snapshot projection, while d is not. Therefore, the assertion on line 17 of Figure 4.11 will observe the following lists:

This representation of the list that starts with element d does not correspond to any state of the program, and is only visible when objects outside of the snapshot projection contain references to objects that are in the projection, creating the illusion of a combined state.

The assertion in Figure 4.11 tries to compare the list d in the post-state with the list a in the pre-state. The e-STROBE framework does not give any protection against such an incorrect combination of states. The program itself is responsible for maintaining the correct state if objects are used that are not in the snapshot projection. A program can do this either through the pre-calculation of results or through the copying of objects, as described in Section 4.3.2.

4.5 Implementation

4.5.1 Snapshots

The STROBE framework associates snapshots with checker threads and uses their identifiers to initialize and use the different snapshots that can co-exists in the framework. The framework uses the Snapshot class to keep track of the snapshots that are active in the program, and uses this information in the object barriers of JikesRVM to check if an object should be copied to preserve its state.

The e-STROBE framework does not associate a snapshot with a checker thread, but uses separate snapshot identifiers to keep track of the active snapshots in the program. These identifiers are generated by the framework when a snapshot is initialized. The e-STROBE framework keeps track of these identifiers in an array, allowing programs to use 1000 snapshots simultaneously. When a snapshot is destroyed, the identifier is deactivated and can be re-issued by the e-STROBE framework for a future snapshot. The limit of 1000 snapshots is chosen arbitrarily. The e-STROBE can be compiled with a different limit if needed.

All objects in the e-STROBE framework will maintain a forwarding array for their snapshot copies, just like the STROBE framework. The e-STROBE framework uses a forwarding array that can contain up to 1000 object copies (one for every snapshot), compared to 12 (one for every checker thread) in the STROBE framework. Therefore, it will consume a little more memory per object.

For every snapshot, the framework keeps a list of objects that have a snapshot copy for the snapshot. This list is used to remove the copies from the forwarding array when a snapshot is destroyed, preventing that the copies are associated with a new snapshot when the snapshot identifier is re-issued, and allowing the garbage collector to clean up the copies that are no longer used.

4.5.2 Barriers

Like the STROBE framework, the e-STROBE framework uses the object barriers in JikesRVM to select the correct object copy when a field of an object is read, and to check if an object copy has to be made when an object is modified. The e-STROBE framework uses the snapshot identifiers instead of the checker identifiers to perform this check. The rest of the barrier implementation is unchanged.

The object read barriers in the STROBE framework are only used for methods with the @ConcurrentCheck annotation that are executed by checker threads, and are therefore not enabled for other threads in the program. The e-STROBE framework still uses this annotation, but enables the read barriers for methods with the @ConcurrentCheck annotation on all threads in the program.

4.5.3 Asynchronous assertions

Checker threads in the e-STROBE framework have both a *checker identifier* and a *snapshot identifier*. The checker identifier is no longer used to identify the snapshot, but is still maintained for logging purposes.

The evaluation of an asynchronous assertion is modified to make use of the new snapshot identifiers. Evaluation is performed in two steps: the program thread prepares the asynchronous assertion for execution and wakes the checker thread, then the checker thread executes the assertion.

The program thread will initialize the snapshot during the preparation and receives its snapshot identifier. It then switches the checker thread to the scope of the snapshot that was just created, and wakes up the checker thread. Figure 4.14a shows this process of waking up a checker thread in pseudo-code.

The checker thread executes the assertion, and switches back to the live scope afterwards. The result of the assertion is then saved in the CHAFuture, giving the program access to the result. The snapshot is destroyed and the checker thread wakes up the program threads that are waiting for the result. Figure 4.14b shows the pseudo-code for this checking process.

4.6 Testing snapshots in multithreaded programs

The previous sections of this chapter describe how snapshot scopes in the e-STROBE framework are used to protect assertions against assertion interference. We give an example of an assertion that is protected against interference in Figure 4.3. We did not observe an assertion violation during repeated execution of this program in the e-STROBE framework, but this does not prove that the framework works for multithreaded programs: the framework might affect the timing of the program in such a way that assertion interference does not occur during the execution of the testDisplay() method.

Therefore, we test the behaviour of the e-STROBE framework in multithreaded programs with a program that synchronizes two threads in such a way that it ensures assertion interference

```
1  public void startMe(CHAFuture future)
2  {
3    CheckerThread checker = waitForFreeChecker();
4    checker.checkToDo = future;
5    checker.snapshotId = Snapshot.initiateProbe();
6    checker.wakeUp();
7  }
```

(a) Waking up a checker thread

```
1  public void run(){
2    while (true) {
3      waitForAsynchronousAssertion();
4      checkToDo.result = checkToDo.compute();
5
6      int snapshotId = this.snapshotId;
7      snapshotId = null;
8
9      Snapshot.completeProbe(snapshotId);
10     checkToDo.notifyAll();
11     checkToDo = null;
12   }
13 }
```

(b) Executing an asynchronous assertion

Figure 4.14: Executing an asynchronous assertion

in the program. We can check that the framework behaves as we specified if this interference does not occur if we execute the same assertion in a snapshot scope.

Figure 4.15 and 4.16 show a new version of the Timer and Display classes that uses a CyclicBarrier to synchronize the Timer and Display thread in such a way that the timer is incremented during every execution of the getTime() method of the Display class. This method is used twice by the assertion on line 24 of Figure 4.16, ensuring that an assertion violation occurs if the assertion is evaluated in the live scope of the program.

This assertion violation is not observed when the assertion is placed in a snapshot scope, as shown in the example. This demonstrates that snapshot scopes correctly preserve the associated snapshot projection of the heap in multithreaded programs.

Similarly, the BrokenDisplay class in Figure 4.4 will never trigger an assertion violation if the timer in Figure 4.15 is used. However, this display is obviously broken. Again, adding a snapshot scope to the testDisplay method, as shown in Figure 4.17, removes the assertion interference and causes the assertion to fail every time.

The snapshot tests in this section use assertions that are not pure: the Timer will behave differently in a program executed with assertions and a program executed without them. We normally do not allow assertions to change the behaviour of the program. However, this incorrect use of assertions gives us an easy way to check that snapshot scopes in the e-STROBE framework will indeed protect assertions against assertion interference.

```
1  public class Timer extends Thread {
2
3      private Integer time = 0;
4      private CyclicBarrier barrier;
5
6      public Timer(){
7          barrier = new CyclicBarrier(2);
8      }
9
10     @ConcurrentCheck
11     public int getTime(){
12         try {
13             barrier.await(); // start timer
14             barrier.await(); // wait for timer to finish
15             return time;
16         } catch (InterruptedException e) {
17             System.out.println("Barrier interrupted.");
18         } catch (BrokenBarrierException e) {
19             System.out.println("Barrier broken.");
20         }
21         return 0;
22     }
23
24     @Override
25     public void run(){
26         try {
27             barrier.await(); // wait for Display.getTime()
28             this.time = this.time + 1;
29             barrier.await(); // notify Display.getTime()
30             barrier.await(); // wait for Display.getTime()
31             this.time = this.time + 1;
32             barrier.await(); // notify Display.getTime()
33         } catch (InterruptedException e) {
34             System.out.println("Barrier interrupted.");
35         } catch (BrokenBarrierException e) {
36             System.out.println("Barrier broken.");
37         }
38     }
39 }
```

Figure 4.15: Timer synchronized with cyclic barriers

```
1  public class Display {
2      private Timer timer;
3
4      public Display(Timer timer){
5          this.timer = timer;
6      }
7
8      @ConcurrentCheck
9      public int getTime(){ return timer.getTime(); }
10
11     @ConcurrentCheck
12     public static void testDisplay(){
13         Timer t = new Timer();
14         t.start();
15
16         Display d1 = new Display(t);
17         Display d2 = new Display(t);
18
19         RVMThread currentThread = RVMThread.getCurrentThread();
20
21         int snapshotId = Snapshot.initiateProbe();
22         currentThread.snapshotId = snapshotId;
23
24         assert d1.getTime() == d2.getTime();
25
26         currentThread.snapshotId = -1;
27         Snapshot.completeProbe(snapshotId);
28     }
29 }
```

Figure 4.16: Display class with synchronized getTime() calls

```
1  public class BrokenDisplay extends Display{
2
3    public BrokenDisplay(Timer timer){ super(timer); }
4
5    @ConcurrentCheck
6    public int getTime(){ return super.getTime() - 1; }
7
8    @ConcurrentCheck
9    public static void testDisplay(){
10     Timer t = new Timer();
11     t.start();
12
13     Display d1 = new Display(t);
14     Display d2 = new BrokenDisplay(t);
15
16     RVMThread currentThread = RVMThread.getCurrentThread();
17
18     int snapshotId = Snapshot.initiateProbe();
19     currentThread.snapshotId = snapshotId;
20
21     assert d1.getTime() == d2.getTime();
22
23     currentThread.snapshotId = -1;
24     Snapshot.completeProbe(snapshotId);
25   }
26 }
```

Figure 4.17: Deterministically creating a false negative with a broken display

4.7 Limitations

The e-STROBE framework has some limitations, that prevent the use of the framework in certain situations. The limitations on the use of native types and static types are directly inherited from the STROBE framework, while the limitations regarding initialization and state transitions, described in Section 3.5.3 and 3.5.4 are solved by the e-STROBE framework. The other known limitations are described in this section.

4.7.1 The ConcurrentCheck annotation

The STROBE framework makes use of the @ConcurrentCheck annotation to determine whether read barriers should be compiled into a method. If a checker thread will use a method with this annotation, object access uses the snapshot projection associated with the assertion, instead of the live state of the heap.

Every method that wants to use a snapshot projection therefore needs a @ConcurrentCheck annotation. This limits the use of the framework: as soon as a method call is made to a method without this annotation, the results are no longer defined.

While the STROBE framework clearly defined the methods that used snapshots, our framework allows for every method on every thread to create snapshots and switch freely between them. We have therefore tried to replace the @ConcurrentCheck annotation with a different strategy to determine whether read barriers should be enabled when a method is executed. This strategy enables read barriers if:

- The VM is running (not compiling)

- The current thread is not a daemon thread (e.g. the garbage collector) or:
- The current thread has a checkerId of 0 or more (the thread is an asynchronous checker thread)

Our implementation of this new strategy broke several locking techniques that are used in Java. Locking is important in multithreaded programs, and not supporting this would conflict with the goal of this research. The @ConcurrentCheck annotation is therefore still used in the e-STROBE framework. A different barrier selection strategy will be left as future work.

4.7.2 Supporting Java 7

The STROBE framework does not support Java 7. OpenJML, the runtime assertion checker that we selected for our case study in the next section, only supports Java 7 and higher. JikesRVM is therefore patched to ignore the version number of Java 7 programs and treat them as if they were Java 6 programs. This change allows the programs that we use to run fine, but users of the e-STROBE framework need to be aware that they may encounter problems if their programs use Java 7 features.

4.8 Conclusion

The e-STROBE framework allows programs to evaluate statements against a snapshot projection of the heap, protecting both synchronous and asynchronous assertions against assertion interference. Furthermore, it allows a program to combine snapshots, providing the protection for assertions that evaluate properties on state transitions. The framework gives programs the freedom to postpone the evaluation of snapshots to a later time in the program, allowing the use of the post-state of the program during the evaluation of the pre-state. This extends the expressiveness of assertions in a program.

With the e-STROBE framework, the program itself is responsible for the management and use of snapshots. Protecting assertions in a multithreaded program is therefore significantly more work, than writing assertions in a single-threaded program. The next chapters will explore the possibility to use this framework to enhance an existing runtime assertion checker for Java to automatically make use of snapshots, allowing programs to write assertions in a multithreaded program with the same ease as writing them in a single-threaded program.

Chapter 5

OpenJML

The e-STROBE framework creates an environment in which we can evaluate assertions without the risk of assertion interference. However, manually creating and using snapshots in a multi-threaded program requires a lot of work, compared to the simple assertions that we can write for single-threaded programs. We will therefore explore the possibility to have a runtime assertion checker automatically encapsulate assertions in snapshot scopes.

We have developed such a strategy for a Java runtime assertion checker named OpenJML [2]. Assertions for this assertion checker are written in the Java Modeling Language (JML) [22]. OpenJML uses its own Java-compiler that translates assertions written in JML to Java expressions, and compiles them into assertions in the Java program.

This chapter will give an overview of JML, and show examples of assertions written in JML for both single-threaded and multithreaded programs. We will show how OpenJML compiles these assertions into a program, and how these compiled assertions are subject to assertion interference.

We will modify the OpenJML runtime assertion checker to use snapshots from the e-STROBE framework in Chapter 6, and show that these snapshots can be used to automatically rule out assertion interference in a multithreaded runtime assertion checker.

5.1 The Java Modeling Language

JML provides a separate specification language for Java programs, that can be used to describe the behaviour of a program. Specifications for this language are placed in special *annotations*, written on a line that start with //@ or enclosed in a block that starts with /*@ and ends with @*/. A normal Java compiler will treat these annotations as comments, but a JML runtime assertion checker will recognize them as parts of a JML specification, and can compile them into assertions that check the correct use of the program. JML specifications are based on the Design by Contract style [23] and can be used for many purposes besides runtime assertion checking, such as documentation and static verification. Burdy et al. [14] provide an extensive description of the many different tools and applications of JML.

A JML runtime assertion checker does not require a specification to describe all aspects of an the entire program. An assertion checker will check the parts that are described by JML and execute the rest of the program as is. This enables an assertion checker to perform checks on programs that make use of libraries without a JML specification, and therefore without any JML assertions.

JML can be used as a specification language for both single-threaded and multithreaded programs. The specification of a multithreaded program has to account for program interference. In general, this means that the program can guarantee less: a timer that is used by one thread can guarantee that its value is increased by exactly one after incrementation, while a timer that is incremented by multiple threads can only guarantee that its value is increased.

The Java Modeling Language is a large language, that contains features to specify the entire Java language. The feature set of JML is partitioned into language levels. Level 0 defines the basic features of JML, such as preconditions, postconditions, invariants and model variables. All tools that claim to support JML should supported these features. Higher levels define features that are used less frequently, and can be implemented by a JML tool afterwards.

The OpenJML runtime assertion checker that we use in this thesis implements all level 0 features and a large part of the features of other language levels. Our changes will only support method specifications of language level 0. We will therefore only describe the features of this basic language level in this chapter.

5.2 Writing a JML specification

We will describe the basic functionality and syntax of JML by means of two examples: a JML specification for the Node class in Figure 3.1 and a specification of the Timer class in Figure 3.8. These specifications are shown in respectively Figure 5.1 and Figure 5.2. The Node class contains a specification that is only correct for single-threaded executions, while the specification of the Timer describes the behaviour of the timer in both single-threaded and multithreaded environments.

5.2.1 Preconditions and postconditions

The behaviour of a method can be described in JML with preconditions and postconditions, using the **requires** and **ensures** keywords. These conditions are written as expressions in the normal Java syntax, with some specific JML extensions. Figure 5.1 shows preconditions on lines 9 and 20 and postconditions on lines 10, 11, 12 and 21.

JML defines extra operators that can be used in JML expressions. The **\old** operator in postconditions, for example, is used to evaluate part of the expression in the pre-state of the method. The result of a method is available in the postcondition through the **\result** keyword. The Node class uses the **\old** operator on line 11 and the **\result** keyword on line 21.

5.2.2 Public and private specifications

A JML specification can be checked in the method itself and when the method is called. However, a method caller can only check the parts of the specification that are visible in its scope. Therefore, method specifications in JML will by default have the same scope as the method they specify, making the specification visible to both the method itself and its caller. A specification can not use features of the class that are not visible in this scope.

Sometimes, a private variable is needed in the public specification of a method. For this purpose, JML allows a private variable to be declared **spec_public**. This addition will allow the variable to be treated as a private variable by the implementation, but as a public variable in the specification of the program. The next property has this annotation, and is used in the specification of the contains method of the Node class.

```
1  public class Node {
2    private /*@ spec_public nullable */ Node next;
3
4    //@ public model int length;
5    //@ represents length <- getLength();
6
7    //@ public ghost Integer calculatedLength = 1;
8
9    /*@ requires !contains(node);
10     @ ensures contains(node);
11     @ ensures length == \old(length) + 1;
12     @ ensures calculatedLength == length;
13     @*/
14   public void addNode(Node node) {
15     node.next = this.next;
16     this.next = node;
17     //@ set calculatedLength = next == null ? 1 : next.calculatedLength + 1;
18   }
19
20   /*@ requires node != null;
21     @ ensures \result == (next == node) ||
22     @  (next != null <==> next.contains(node));
23     @*/
24   public /*@ pure */ boolean contains(Node node){
25     if(this.next == null)
26       return false;
27     else
28       return this.next == node || this.next.contains(node);
29   }
30
31   public Integer getLength(){
32       return next == null ? 1 : next.getLength() + 1;
33   }
34 }
```

Figure 5.1: JML specification of the Node class

5.2.3 Pure methods

We stated in Chapter 2 that assertion should be transparent: a program that is executed with assertions must behave exactly like a program that is executed without. To ensure this transparency, OpenJML restricts the use of method calls in its specifications to pure methods. Such methods can be defined in JML by adding the /*@ **pure** */ annotation to a method declaration. The contains method in Figure 5.1 has this annotation, and can therefore be used in the specification of the addNode method.

5.2.4 Null value

The /*@ **non_null** */ annotation in a field declaration is used to indicate that a field is not allowed to have the **null** value assigned to it. OpenJML enables this annotation by default, disallowing null values for all class and instance fields. The /*@ **nullable** */ annotation is used to indicate that the next property may be **null**. The next field on line 2 of Figure 5.1 has this annotation. Otherwise, we would not be able to end the linked list.

5.2.5 Model and ghost fields

Model fields are used to define extra fields in a class that are only used for specification. These fields are declared using the **model** keyword. The value of this field is specified with the **represents** clause. This clause is used to assign an expression to the model field, that will be used to calculate its value. Lines 4 and 5 of Figure 5.1 specify the model field length representing the length of the list, including the node that calculates it.

A different way to specify extra properties in a class, is through the use of ghost fields. These fields are declared with the **ghost** keyword, and are also only used in specifications. These fields act as normal instance fields. They are initialized with a value, and can be updated with the **set** clause, placed in a JML comment. Figure 5.1 defines the length of a list in a ghost field, on line 17. Note that ghost values are set programmatically, and therefore do not automatically represent the correct value. In fact, the calculatedLength field can incorrectly represent the length of the list if the following list is built:

```
Node a = new Node();
Node b = new Node();
Node c = new Node();
a.addNode(b);
b.addNode(c);
```

The calculatedLength field of Node a will now (incorrectly) report a length of 2, while the list has a length of 3. As ghost fields are updated programmatically, it is the responsibility of the programmer to ensure that ghost fields correctly represent the property they model.

5.2.6 Invariants and constraints

Class invariants define properties of a class that are true throughout the execution of the program. These properties are defined with the **invariant** keyword. The Timer in Figure 5.2, like most timers, will never have a negative time value. This property is specified by the invariant on line 4.

History constraints specify how a class changes during the execution of the program, using a **constraint** clause. Every modification of the class must satisfy the constraint defined by the expression in this clause. A constraint will typically use a combination of two states: one for the state before a modification and one for the current state of the program. The **old** keyword

```
1  public class Timer {
2      private /*@ spec_public @*/ Integer time = 0;
3
4      //@ public invariant time >= 0;
5      //@ public constraint time >= \old(time);
6
7      /*@ ensures \result >= \old(time) && \result <= time;
8        @*/
9      public /*@ pure @*/ int getTime(){ return time; }
10
11       @ ensures getTime() > \old(getTime());
12       @*/
13     public void increase(){
14         time = time + 1;
15     }
16
17     public static void main(String[] args){
18         Timer timer = new Timer();
19         timer.increase();
20         timer.increase();
21         timer.getTime();
22     }
23 }
```

Figure 5.2: JML specification of the Timer class

is used in these expressions to refer to the state before the modification of the class. Line 5 of Figure 5.2 defines a constraint on the Timer class, that specifies that the time of this timer may only increase during the execution of the program.

5.3 Translating a JML specification with OpenJML

The OpenJML compiler translates the JML specification of a program into assertions that check if the program behaves according to its specification. In this section we will describe how this translation is performed, and which checks are added to the program by OpenJML. We will describe the OpenJML compiler using the specification of the Node and Timer classes that we have given in the previous section.

The OpenJML User Guide [17] gives an overview of the use of OpenJML. However, the translation that is performed by the OpenJML compiler is not completely described in this document. The translated programs given in this section are therefore obtained with the use of CFR [3] a Java decompiler. The programs in Figure 5.3 and 5.4 are simplified for readability.

5.3.1 Preconditions and postconditions

OpenJML checks the preconditions and postconditions of a method respectively before and after executing the statements of the method body. The **requires** clauses of a method specification are translated into assertions that are executed at the beginning of the method and the **ensures** clauses are translated into assertions at the end of the method. The **requires** clause on line 9 of Figure 5.1, for example, is translated to the assertion on line 7 of Figure 5.3, and the **ensures** clause on line 10 of Figure 5.1 is translated into the assertion on line 13 of Figure 5.3.

OpenJML checks the specification of a method twice: in the method itself and in the method body of the caller. This enables OpenJML to check properties on libraries that are not compiled

```
1  public class Node {
2    public Node next;
3    public Integer calculatedLength = 1;
4
5    public void addNode(Node node) {
6      int oldLength = _JML$model$length();
7      assert !_JML_METHODEF__contains(node);
8
9      node.next = this.next;
10     this.next = node;
11     calculatedLength = next == null ? 1 : next.calculatedLength + 1;
12
13     assert _JML_METHODEF__contains(node);
14     assert _JML$model$length() == oldLength + 1;
15     assert calculatedLength ==  _JML$model$length();
16   }
17
18   public boolean contains(Node node){
19     assert node != null;
20
21     boolean result;
22     if(this.next == null)
23       result = false;
24     else
25       result = this.next == node || this.next._JML_METHODEF__contains(node);
26
27     assert result == (next == node) ||
28       (next != null == next._JML_METHODEF__contains(node));
29     return result;
30   }
31
32   private boolean _JML_METHODEF__contains(Node node) {
33     assert node != null;
34
35     boolean result = contains(node);
36
37     assert result == (next == node)
38       || (next != null == next._JML_METHODEF__contains(node));
39     return result;
40   }
41
42   public Integer getLength(){
43     return next == null ? 1 : next.getLength() + 1;
44   }
45
46   public int _JML$model$length(){
47     return getLength();
48   }
49 }
```

Figure 5.3: Translated JML specification of the Node class

with the OpenJML compiler. OpenJML builds a wrapper method for the checks in the class that calls the method. This wrapper will first check the preconditions, then execute the method, and then execute the postconditions. A wrapper of the contains method of the Node class is shown on line 32 - 40 of Figure 5.3. All method calls to the contains method are replaced by a method call to this wrapper, except for one actual method call that is done by the wrapper.

The **\result** keyword in the postcondition of a method represents the return value of a method. All return statements in the method are therefore replaced by a local result variable, followed by a jump to the end of the method. The **\result** keyword in the postconditions is replaced by a reference to this local variable during the translation of the postcondition expressions. This process is shown in Figure 5.3 on line 21, 23 and 25.

An **\old** expression is used in OpenJML in an expression that checks the state transition between the pre-state and the post-state of a method. OpenJML calculates these expressions like we did manually in Section 4.3.1: the **\old** expression is pre-calculated in the pre-state of the method, and the result of this precalculation is saved in a local variable. The **\old** expressions in the postcondition are then replaced by a reference to this variable. This process transfers the relevant parts of the pre-state to the postcondition, allowing OpenJML to check the properties of the state transition that is caused by the execution of the method.

5.3.2 Public and private specifications

The OpenJML compiler will treat class fields with a **spec_public** annotation as public fields, and compiles them as such. This allows every part of the specification to access these variables. Both the next field of the Node class and the time field of the Timer class are therefore public after compilation.

5.3.3 Model and ghost fields

The OpenJML compiler replaces model fields with the expression that is given in the **represents** clause. Therefore, every use of the length field in Figure 5.3 is replaced by a getLength() method call. Ghost fields, on the other hand, are translated into normal fields. The **set** statements that modify the values of ghost fields become assignments in the compiled program.

5.3.4 Pure methods

Pure annotations are used by the syntax checker of OpenJML. The OpenJML compiler adds a @Pure annotation to these methods, but does not check the pureness of a method at runtime.

5.3.5 Invariants and constraints

Class invariants must hold in every visible state of the program. OpenJML chooses to check these invariants whenever preconditions or postconditions are checked. This not only results in invariant checks for the increase method of Figure 5.4 on line 18 and 25, but also for invariant checks when the getTime() method is called, on line 20.

History constraints must also hold between two consecutive visible states of a program. Complex constraints can be invalidated by every modification of any class. However, checking all history constraints in a program after every modification is impossible. OpenJML therefore treats these constraints as extra postconditions, and checks them whenever the postconditions of a method are checked.

```
1  public class Timer {
2    public Integer time = 0;
3
4    public int getTime(){
5      assert time >= 0;
6      Integer oldtime = time;
7
8      Integer result = time;
9
10     assert time >= oldtime;
11     assert time >= 0;
12     assert result >= oldtime && result <= time;
13
14     return result;
15   }
16
17   public void increase(){
18     assert time >= 0;
19     Integer oldtime = time;
20     int oldgetTime = _JML_METHODEF__getTime();
21
22     time = time + 1;
23
24     assert getTime() > oldgetTime;
25     assert time >= 0;
26     assert time >= oldtime;
27   }
28
29   public int _JML_METHODEF__getTime(){
30     assert time >= 0;
31     Integer oldtime = time;
32
33     int result = getTime();
34
35     assert time >= oldtime;
36     assert time >= 0;
37     assert result >= oldtime && result <= time;
38
39     return result;
40   }
41 }
```

Figure 5.4: Translated JML specification of the Timer class

5.4 Using OpenJML in multithreaded programs

Section 4.2 showed an example of a Timer that is simultaneously displayed on two Displays. This example is subject to assertion interference: the program behaves exactly as specified, but modification of the Timer during assertion checking leads to the false report of an assertion violation. We will show in this section that the same situation can occur when we use the OpenJML runtime assertion checker in a multithreaded environment.

Figure 5.5 shows a JML annotated Timer and Display. The two Displays in this example are attached to the Timer using the DisplaySet class, instead of the testDisplay method. The assertion in the testDisplay has been converted to the JML postcondition on line 33 of Figure 5.5. OpenJML will translate this specification to assertions in the method body. According to the previous section, this will result in the following (simplified) translation:

```
public DisplaySet(Timer t){
  assert t != null;
  d1 = new Display(t);
  d2 = new Display(t);
  assert d1.getTime() == d2.getTime();
}
```

The Timer can increase its value during the execution of the assertion, leading to an assertion violation similar to the assertion violation that we encountered in the testDisplay method in Figure 4.2. As we have seen in Section 4.2, the assertion violation is caused by assertion interference, and does not represent a fault in the program. The OpenJML runtime assertion checker is therefore not trustworthy if it is used for the checking of multithreaded programs.

Analogous to Section 4.2, OpenJML might fail to detect an assertion violation if the DisplaySet is initialized with the BrokenDisplay on line 48 of Figure 5.5:

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

The assertion violation in this example is present in every possible state of the program, and OpenJML should therefore report it in every execution of the program.[1]

5.5 Implementation of the runtime checker

The next chapter will use the e-STROBE framework to prevent assertion interference in the OpenJML runtime assertion checks. This solution requires modifications to the OpenJML compiler. We will therefore describe the design of this compiler and explain how it adds the associated assertions to the parts that we will modify: method specifications and method calls.

5.5.1 Design of the OpenJML compiler

OpenJML is built on top of the OpenJDK Java compiler [5]. This Java compiler is built in Java, and contains the standard components of a compiler [29]: a scanner and parser for syntactical

[1] The execution of this example failed, due to an error in OpenJML: **super**.getTime() − 1 was wrongfully compiled into **this**.getTime() − 1. This error has been reported to the OpenJML developers.

```
1  public class Timer extends Thread {
2      private Integer time = 0;
3
4      public /*@ pure */ int getTime(){
5          return time;
6      }
7
8      @Override
9      public void run(){
10         for(int i = 0; i < 10000; i++){
11             time = time + 1;
12         }
13     }
14 }
15
16 public class Display {
17     private Timer timer;
18
19     public Display(Timer timer){
20         this.timer = timer;
21     }
22
23     public /*@ pure */ int getTime(){
24         return timer.getTime();
25     }
26 }
27
28 public class DisplaySet {
29     private /*@ spec_public */ Display d1;
30     private /*@ spec_public */ Display d2;
31
32     /*@ requires t != null;
33       @ ensures d1.getTime() == d2.getTime();
34       @*/
35     public DisplaySet(Timer t){
36         d1 = new Display(t);
37         d2 = new Display(t);
38     }
39
40     public static void main(String[] args){
41         Timer t = new Timer();
42         t.start();
43
44         DisplaySet d = new DisplaySet(t);
45     }
46 }
47
48 public class BrokenDisplay extends Display {
49     public BrokenDisplay(Timer timer){
50         super(timer);
51     }
52
53     public /*@ pure */ int getTime(){
54         return super.getTime() - 1;
55     }
56 }
```

Figure 5.5: JML specification that is subject to assertion interference

analysis, several compilation cycles that perform step-by-step contextual analysis on the abstract syntax tree (AST), and a code generator that desugars the decorated abstract syntax tree into the bytecode for the .**class** files of the program. These components are built in such a way that the behaviour of the compiler can be modified through the extension of the associated Java classes.

OpenJML uses this extensibility to modify several components of the compiler [18]. The scanner is extended with rules to recognize JML comment blocks. The parser converts these blocks into JML specification nodes in the abstract syntax tree (AST). These nodes are then visited by the JML tree parser during contextual analysis, and translated to the associated assertion statements in the decorated AST. These assertion statements are regular Java statements, and will be converted into bytecode by the OpenJDK code generator.

5.5.2 Adding specifications to the AST

OpenJML extends the lexical scanner and parser with the ability to recognize JML comment blocks. The scanner parses extra keywords within these blocks to support the JML syntax, including **requires**, **old** and **ensures**. Parsing these comment blocks will produce subtrees that can be added to the abstract syntax tree of the program.

The JML specifications are added to the node that they describe: preconditions and postconditions are added to the method that they belong to, and invariants, model fields and history constraints are added to the class that they are defined for. The OpenJDK abstract syntax tree does not provide the possibility to add these extra nodes to the methods and classes. Therefore, the OpenJML compiler extends the OpenJDK nodes with its own version that provides the possibility to collect these specifications. A method declaration, saved in the JCTree.JCMethodDecl node in OpenJDK is specialized in OpenJML into a JMLTree.JMLMethodDecl, and a class, saved in the JCTree.JCClassDecl is specialized into a JMLTree.JMLClassDecl.

5.5.3 Converting the specification to assertions

OpenJML inserts an extra step into the context analysis compilation phase of OpenJDK that translates the OpenJML AST nodes into the program statements that are needed to check the specification. This step will use a tree walker to visit the entire syntax tree, collect all the specifications that apply to a certain part of the program, and build the statements that check this specification.

When a JMLTree.JMLMethodDecl is visited, the tree walker will collect all the preconditions, invariants and **old** expressions and insert them as extra statements at the beginning of the method, as described in Section 5.3. Similarly, postconditions, invariants and history constraints are inserted at the end of the method.

A method call, identified by the JMLTree.JMLMethodInvocation node, is replaced by a method call to a newly created method that is placed in the caller's class. This method will wrap the actual method call into precondition and postcondition checks, as described earlier in Section 5.3.

5.6 Conclusion

JML allows a program to reason about the behaviour of the program, instead of the checks that are needed to check that behaviour. The OpenJML runtime checker automatically compiles a JML specification into statements that check if the execution of a program behaves correctly according to its JML specification, and report if it behaves differently.

OpenJML checks the correct execution of a method by means of statements that are placed at the beginning and at the end of the method body. These statements check that a method is called under the right circumstances, and that the execution of the method body produces the correct result.

JML specifications can be used in both single-threaded and multithreaded programs. OpenJML checks in multithreaded programs, however, are subject to assertion interference.

Chapter 4 provides a solution for assertion interference that requires assertions to be placed in snapshot scopes. We need to place these scopes around the OpenJML checks to have the same protection in OpenJML. The next chapter will extend the OpenJML compiler to use these snapshot scopes for the checking of method specifications.

Chapter 6

Multithreaded OpenJML

OpenJML converts specifications into assertions, allowing the user to focus on the behaviour of the program, instead of on the checking. The conversion, however, does not take assertion interference into account. This makes OpenJML untrustworthy for multithreaded programs: the runtime assertion checker might not detect an assertion that could have been detected, or might report an assertion violation for a failure that is not in the program.

This chapter will discuss the integration of snapshots in OpenJML. The e-STROBE framework will be used to protect OpenJML assertions from assertion interference. The integration will be done in the OpenJML compiler, and requires no changes to the specification. We will present this integration as a proof of concept. Therefore, we will only use it to protect the checks that are performed in the pre-state and post-state of a method. However, the techniques that are used in this proof of concept will be universal, and can be applied to the rest of OpenJML in a future project.

We will begin this chapter with an analysis of the OpenJML assertions that need to be protected. We will discuss when snapshots need to be taken and which statements should be executed in a snapshot scope. We will show that the use of snapshot scopes will not change the behaviour of OpenJML assertions, apart from the exclusion of assertion interference,

We will then present the changes that we made to the OpenJML compiler. We will check that these changes will indeed protect the OpenJML assertions, both through decompilation and through the execution of a program with locks, that deterministically triggers assertion interference.

Finally, we will discuss to what extend we have protected OpenJML from assertion interference, and give an overview of the limitations of our solution, and the work that still needs to be done.

Build and usage instructions for the e-STROBE OpenJML extensions can be found in Appendix B. This appendix also contains the location to the source code and binaries for these extensions.

6.1 Protecting OpenJML with snapshots

OpenJML checks the behaviour of a method by placing checks at the beginning and end of a method. Section 5.3 shows that these checks are not limited to the preconditions and postconditions of the method, but will also contain invariants and history constraints. All these checks can use objects on the heap, and are therefore subject to assertion interference in multithreaded programs.

```
1  public void addNode(Node node) {
2      RVMThread currentThread = RVMThread.getCurrentThread();
3
4      int preId = Snapshot.initiateProbe();
5      currentThread.snapshotId = preId;
6      int oldLength = _JML$model$length();
7      assert !_JML_METHODEF__contains(node);
8      currentThread.snapshotId = -1;
9      Snapshot.completeProbe(preId);
10
11     node.next = this.next;
12     this.next = node;
13     calculatedLength = calculatedLength + 1;
14
15     int postId = Snapshot.initiateProbe();
16     currentThread.snapshotId = postId;
17     assert _JML_METHODEF__contains(node);
18     assert _JML$model$length() == oldLength + 1;
19     assert calculatedLength ==  _JML$model$length();
20     currentThread.snapshotId = -1;
21     Snapshot.completeProbe(postId);
22 }
```

Figure 6.1: Protecting OpenJML checks with snapshots

We have modified OpenJML to use snapshots to protect these checks. We have done this by placing snapshots scopes, as explained in Section 4.1: one scope around all the OpenJML generated statements in the pre-state of a method and one around all the statements in the post-state of a method. Figure 6.1 demonstrates the use of snapshots to protect the addNode method in Figure 5.3. The snapshots that we use in OpenJML are always created in the statement before entering a snapshot scope, and destroyed in the statement after switching back. We will discuss the placement of these snapshot scopes and check if all possible uses of local variables satisfy the constraints that we described in Section 4.4.6.

6.1.1 Placement of snapshot scopes

In this prototype, we will consider preconditions, postconditions, invariants, history constraints, model fields and ghost fields. The expressions of preconditions, postconditions, invariants and history constraints can all contain instance fields. These fields may be accessed by all threads, and the accesses to these fields in OpenJML thus needs to be performed inside of a snapshot. Note that all these expressions in OpenJML are pure. Therefore, these statements can be executed in a snapshot scope. Placing these statements in a snapshot scope does not change the behaviour, apart from the assertion interference that might have occurred otherwise.

Model fields in OpenJML are replaced by the code they represent. This code is therefore executed in the snapshot scope. The expression that represents a snapshots scope is pure, and we can therefore safely execute it in the snapshot scope.

Ghost fields use two types of JML clauses: the initialization of the field and the **set** statements to update the field. Both the initialization and the **set** statements are not executed in the pre-state or post-state of a method: the initialization is performed when the object is created and the **set** statements are placed inside the method bodies of a class. These statements are therefore not evaluated in the snapshot scopes of the pre-state or post-state. However, the expressions of both **set** statements and the initialization of the field are pure, and we can evaluate them atomically

```
1  public class ArrayExample {
2      private /*@ spec_public */ Integer[] someArray;
3
4      public ArrayExample(){
5          someArray = new Integer[2];
6          someArray[0] = 1;
7          someArray[1] = 4;
8      }
9
10     //@ ensures \old(someArray[\result]) == 4;
11     //@ ensures \old(someArray[\result].intValue()) == 4;
12     //@ ensures \old(someArray[\result] + 2) == 6;
13     //@ ensures \old(someArray[\result]) + 2 == 6;
14     public int someMethod(){
15         return 1;
16     }
17
18     public static void main(String[] args){
19         ArrayExample example = new ArrayExample();
20         example.someMethod();
21     }
22 }
```

Figure 6.2: JML specification of the Node class

if we place them in snapshot scopes, removing the need of locking around these statements. Exploring this possibility, and discussing the correctness of it, is left for future work.

6.1.2 State transitions in OpenJML

Our snapshot implementation for OpenJML will always take a snapshot, and immediately switch to it. Therefore, the local variables that are used in a snapshot scope will never be changed between taking a snapshot and entering the associated snapshot scope.

The only state transitions in OpenJML that are within the scope of this project, are **\old** expressions. The evaluation of these expressions is placed in the snapshot scope of the pre-state, while the result of this expression is used in the post-state.

The **\old** expression is evaluated before the creation of the local variables of the method body, and can therefore not use these variables. Therefore, the only local variables that are used in a snapshot scope are in the post-state, and contain the pre-calculated result of the **\old** expressions. The snapshot of the post-state is created after the last modification of these variables. This particular use of local variables in snapshot scopes is therefore correct, and will not change the behaviour of the checks or the program, apart from the timing.

A special local variable that is used in the program itself, and in the snapshot scopes is the result of the method. This variable is only changed during the execution of the program, and is only valid after the execution of the program body. However, the use of this keyword in **\old** expressions can trigger the use of this field in the pre-state, as shown in Figure 6.2. This example contains an array, and uses the **\result** keyword to check the value of a particular element in the array. The assertions on line 11 and 12 of this example will trigger an assertion violation, but the assertions on line 10 and 13 are correct. OpenJML detects the combination of array access and the **\result** keyword, and saves a copy of the entire array. The array element that the **old** expression represents is then selected in the post-state. A simplified OpenJML translation of the method is as follows:

```
1  public int someMethod(){
2    RVMThread currentThread = RVMThread.getCurrentThread();
3    int result;
4    int preId = Snapshot.initiateProbe();
5    currentThread.snapshotId = preId;
6    Integer old2 = someArray[result].intValue();
7    Integer old3 = someArray[result] + 2;
8    Integer[] arrayClone = someArray.clone();
9    currentThread.snapshotId = -1;
10   Snapshot.completeProbe(preId);
11
12   result = 1;
13
14   int postId = Snapshot.initiateProbe();
15   currentThread.snapshotId = postId;
16   assert arrayClone[result] == 4;
17   assert old2 == 4;
18   assert old3 == 6;
19   assert arrayClone[result] + 2 == 6;
20   currentThread.snapshotId = -1;
21   Snapshot.completeProbe(postId);
22
23   return result;
24 }
```

Figure 6.3: JML specification of the Node class

```
public int someMethod(){
  int result;
  Integer old2 = someArray[result].intValue();
  Integer old3 = someArray[result] + 2;
  Integer[] arrayClone = someArray.clone();

  result = 1;

  assert arrayClone[result] == 4;
  assert old2 == 4;
  assert old3 == 6;
  assert arrayClone[result] + 2 == 6;
  return result;
}
```

This shows us why two of the postconditions of someMethod fail: these expressions do not match the specific pattern of an array access with the result as the index, and do not trigger this specialized translations. Therefore, old2 and old3 will have the values 1 and 3, instead of the expected values 4 and 6.

Figure 6.3 shows the snapshot scopes that we place around the OpenJML statements in the pre-state and post-state of someMethod. Two additional local variables are now used in snapshot scopes: arrayClone and result. The result variable is only changed by the program itself, and will never be changed inside the snapshot scopes. Therefore, the use of snapshot scopes will not change the behaviour of these expressions. The arrayClone variable is created in the pre-state, and used in the post-state. The snapshot projection of the post-state includes the arrayClone array. Therefore, the use of snapshots does not change the behaviour of expressions that use this array copy.

```
1  protected JCVariableDecl createAndGoToSnapshot(DiagnosticPosition
2      preferredPosition, ListBuffer<JCStatement> stats) {
3    JCExpression snapshotExpression =
4      methodCallUtilsExpression(preferredPosition,"createSnapshot");
5
6    Name n = names.fromString(Strings.strobeSnapshot + (++count));
7    JCVariableDecl vd = treeutils.makeVarDef(snapshotExpression.type, n,
8      methodDecl.sym, snapshotExpression);
9    stats.add(vd);
10
11   JCExpression ex = treeutils.makeIdent(methodDecl.pos,vd.sym);
12   JCStatement switchCall =
13     methodCallUtilsStatement(preferredPosition, "switchToSnapshot", ex);
14   stats.add(switchCall);
15
16   return vd;
17 }
```

(a) Creating and immediately switching to a snapshot

```
1  protected void switchBackAndDestroySnapshot(DiagnosticPosition preferredPosition,
2      JCVariableDecl vd, ListBuffer<JCStatement> stats) {
3
4    // switch to live
5    JCExpression ex = treeutils.makeIdent(methodDecl.pos,vd.sym);
6    JCStatement createCall =
7      methodCallUtilsStatement(preferredPosition, "switchToLive", ex);
8    stats.add(createCall);
9
10   // destroy snapshot
11   ex = treeutils.makeIdent(methodDecl.pos,vd.sym);
12   JCStatement switchCall =
13     methodCallUtilsStatement(preferredPosition, "destroySnapshot", ex);
14   stats.add(switchCall);
15 }
```

(b) Switching to the live state, and destroying the snapshot

Figure 6.4: AST utility methods for snapshots

6.2 Implementation

Implementing snapshots for OpenJML does not add any syntax to the JML specifications, and therefore only needs changes to the contextual analysis of the OpenJML compiler. During this phase, OpenJML will insert the extra statements that evaluate the preconditions and postconditions of methods.

We will first discuss the helper methods that are added to the compiler to ease the insertion of snapshots in the code. We will then discuss the use of these methods to protect the checks in the method itself, and in the method caller.

6.2.1 Creating helper methods for snapshots

In our OpenJML snapshot implementation, the start of a snapshot scope will always occur immediately after the creation of a snapshot. Therefore, we wrote a helper method, that created the AST nodes to add both these statements to a program. This helper method is shown in

```
1 public static int createSnapshot(){
2   if(RVMThread.getCurrentThread().snapshotId != -1)
3     return -1;
4   return Snapshot.initiateProbe();
5 }
```

(a) Creating a snapshot

```
1 public static void switchToSnapshot(int snapshotid){
2   if(snapshotid == -1)
3     return;
4   RVMThread.getCurrentThread().snapshotId = snapshotid;
5 }
```

(b) Switching to a snapshot

```
1 public static void switchToLive(int snapshotid){
2   if(snapshotid == -1)
3     return;
4   RVMThread.getCurrentThread().snapshotId = -1;
5 }
```

(c) Switching back to the live state of the heap

```
1 public static void destroySnapshot(int snapshotid){
2   if(snapshotid == -1)
3     return;
4   Snapshot.completeProbe(snapshotid);
5 }
```

(d) Destroying a snapshot

Figure 6.5: Runtime utility methods for snapshots

Figure 6.4a. The helper method will return an AST node for the snapshot identifier, that will be used when the program switches back and destroys the snapshot. These two statements are also executed consecutively, and are combined in the switchBackAndDestroySnapshot helper method, shown in Figure 6.4b.

OpenJML expressions can contain method calls. These called methods can themselves contain snapshot scopes, resulting in the creation of new snapshots, inside snapshot scopes. Section 4.4 explains that this will lead to the creation of a new snapshot of the live state of the heap. This will not lead to the intended behaviour of the method call: the statements should be executed on the same snapshot projection as the outer snapshot scope. The nested snapshot calls are therefore unnecessary: heap access is already protected by the outer snapshot scope. We have created four extra snapshot methods, that prevent this nested snapshot switching. These helper methods are shown in Figure 6.5.

With these helper methods, the initialization of a nested snapshot scope will not create a snapshot, and instead return snapshot id -1. This identifier is used in the subsequent switchToSnapshot and switchToLive methods, to prevent the snapshot scope switching that would normally occur. Finally, calling the destroySnapshot method with the -1 identifier prevents snapshot destruction of the non-existent snapshot.

6.2.2 Using snapshots in a method declaration

We can now use the helper method in Figure 6.4 to insert our snapshot statements around the OpenJML checks.

OpenJML uses the addPrePostConditions method in the JMLAssertionAdder class to convert a specification into the checks that are executed in a method body. This method will collect the OpenJML statements in three lists of statements, that are inserted into the method: preStats, ensuresStats and exsuresStats. preStats contains all OpenJML code that is executed in the pre-state of the method, while ensuresStats and exsuresStats contain the statements that are executed when the method finishes respectively normally, or exceptionally. We will insert our snapshot creation statements directly after the initialization of these lists, using the previously described helper methods:

```
JCVariableDecl preVariable = createAndGoToSnapshot(methodPos, preStats);
JCVariableDecl ensuresVariable = createAndGoToSnapshot(methodPos,
    ensuresStats);
JCVariableDecl exsuresVariable = createAndGoToSnapshot(methodPos,
    exsuresStats);
```

This will place our snapshot code before any checks that are generated by OpenJML. The addPrePostConditions method is a method that processes all JML specifications that apply to a method, and adds all the checks that OpenJML performs at the beginning and end of a method. All these check are collected in the three lists, and inserted into the AST of the method body at the end of the addPrePostConditions method. We will add our snapshot destruction statements to this list, just before this insertions takes place:

```
switchBackAndDestroySnapshot(methodPos, preVariable, preStats);
switchBackAndDestroySnapshot(methodPos, ensuresVariable, ensuresStats);
switchBackAndDestroySnapshot(methodPos, exsuresVariable, exsuresStats);
```

These extra statements ensure that the snapshot statements are inserted into the method at the positions that are described in Section 6.1.

6.2.3 Using snapshots in a method call

OpenJML adds its checks when a method is called. We also protect these checks with snapshot scopes. The statements in this method are not built in the addPrePostConditions method, but in the applyHelper method of the JMLAssertionAdder class. Again we insert the snapshot statements at the beginning and the end of the JML statement blocks, analogous to our method body implementation.

6.2.4 Using @ConcurrentCheck annotations

The e-STROBE framework needs @ConcurrentCheck annotations to activate the snapshot scopes in a method. Our implementation will not insert these statements to the methods that use snapshots. Therefore, these statements need to be added manually, when the program is written. Automatically adding these statements to the program should be trivial, but is left as future work.

6.3 Testing the implementation

We have tested our implementation in two ways. First, we have decompiled our examples, to verify that our implementation places the snapshots at the places that we specify in Section 6.1.

Second, we created examples of multithreaded programs that deterministically trigger assertion interference, as we did before in Section 4.6.

6.3.1 Decompiling

The correct placement of the snapshot statements is tested by decompiling the OpenJML examples in this thesis, and checking them by hand. We have decompiled the examples in Figure 5.1, 5.2, 5.5 and 6.2, using the CFR Java decompiler [3]. This shows us that the JML checks are indeed protected from assertion interference in the way that we described in Section 6.1. Decompilation shows that the snapshots scopes that we added in the applyHelper method are not only used for the checking of method calls, but also add other checks to the code, such as checks on runtime exceptions. These checks are therefore also protected by our changes. On first sight, putting these checks in a snapshot scope looks valid, and the correct execution results of our examples seem to support this. However, more testing is needed to confirm the correctness of these scopes.

6.3.2 Testing a multithreaded program

We test the correct operation of snapshots in multithreaded programs with a CyclicBarrier, like we did earlier in Section 4.6. We will use the Timer and Display example in Figure 5.5 as a basis for our test. Just as in Section 4.6, we use the CyclicBarrier in such a way, that each getTime method call will trigger an incrementation of the timer. The modified Timer class for this test is shown in Figure 6.6.

The execution of the checks in this program triggered an assertion violation, if the program is compiled with the normal OpenJML compiler. It no longer triggers this violation if the program is compiled with snapshots and executed in the e-STROBE framework. Similarly, using the BrokenDisplay as described in Section 5.4 should always trigger an assertion violation. However, we could not test this, due to the bug in OpenJML that we described in Section 5.4.

6.4 Conclusion

We have successfully used the e-STROBE framework to protect the OpenJML method checks against assertion interference. We did this in a transparent way: snapshots are added in the OpenJML compiler. Therefore, end users do not need to make any changes to their specifications.

The protection of the method checks in OpenJML has shown that the e-STROBE frameworks protects the assertions of OpenJML. We have created the basis for the safe use of this runtime assertion checker in multithreaded programs. This protection can be extended to the entire OpenJML runtime assertion checker, using the same principles that we used to protect the method checks. We will leave this as future work.

The integration of e-STROBE with OpenJML shows us that snapshots can be encapsulated in a runtime assertion checker, shielding the e-STROBE framework from the user. We have used OpenJML as a proof of concept; other runtime assertion checkers can use the e-STROBE framework for a similar protection of their checks against assertion interference. Therefore, the e-STROBE framework is a generic platform to protect multithreaded runtime assertion checking against assertion interference.

```
1  public class Timer extends Thread{
2      private Integer time = 0;
3      private CyclicBarrier barrier;
4
5      public Timer(){
6          barrier = new CyclicBarrier(2);
7      }
8
9      @ConcurrentCheck
10     public /*@ pure */ int getTime(){
11         try {
12             barrier.await(); // start timer
13             barrier.await(); // wait for timer to finish
14             return time;
15         } catch (InterruptedException e) {
16             System.out.println("Barrier interrupted.");
17         } catch (BrokenBarrierException e) {
18             System.out.println("Barrier broken.");
19         }
20         return 0;
21     }
22
23     @Override
24     public void run(){
25         try {
26             barrier.await(); // wait for Display.getTime()
27             this.time = this.time + 1;
28             barrier.await(); // notify Display.getTime()
29             barrier.await(); // wait for Display.getTime()
30             this.time = this.time + 1;
31             barrier.await(); // notify Display.getTime()
32         } catch (InterruptedException e) {
33             System.out.println("Barrier interrupted.");
34         } catch (BrokenBarrierException e) {
35             System.out.println("Barrier broken.");
36         }
37     }
38 }
```

Figure 6.6: Timer with cyclic barriers in a JML annotated program

Chapter 7

Conclusion

7.1 Summary

We have shown that the evaluation of assertions in multithreaded Java programs can give unexpected results due to assertion interference, caused by the changes that are made by other threads. Assertion interference makes runtime assertion checkers no longer trustworthy: the reported assertion violations do not necessarily resemble faults in the program.

We developed the e-STROBE framework, as an extension to the STROBE framework, to prevent assertion interference. The STROBE framework provided asynchronous assertions to speed up assertion checking in programs. These asynchronous assertions used snapshots to save the state of the program that was used by the assertion. However, the use of these snapshots was limited to asynchronous assertions, preventing the easy integration of snapshots in runtime assertion checking.

The e-STROBE framework allows statements to be executed in snapshots scopes, that redirect heap access in such a way that the program acts as if the statements are executed on the snapshot. We show that snapshots not only can be used to evaluate a single state, but also can be combined to protect state transition checks from assertion interference. Finally, we tested that our implementation works as specified.

We applied these snapshots to OpenJML, to protect the checks that this runtime assertion checker generates against assertion interference. We provide this protection to the assertions that are evaluated in the pre-state and post-state of a method. However, the concepts that are used in this proof of concept are universal, and can be extended to support other features of OpenJML. The full integration of the e-STROBE framework with OpenJML is left as future work.

7.2 Evaluation

We began this research with the following research question:

> *Can we create an environment in which we can perform runtime assertion checking in multithreaded programs without the risk of assertion interference?*

We conclude that such an environment can indeed be created. The application, however, is still limited: the e-STROBE-framework is built on top of JikesRVM, and can therefore not be

used in every Java environment. Therefore, protection against assertion interference in a different virtual machine might require a completely different technique.

The e-STROBE framework is still a prototype, and only implements snapshots for non-static object reference fields. However, this limitation can easily be overcome, by implementing snapshot barriers for the other field types.

The integration of the e-STROBE framework with OpenJML uses this safe environment in such a way that the programmer does not need to worry about snapshots. The programmer can therefore write his specifications in JML, while the protection against assertion interference is completely shielded by the OpenJML compiler.

The applicability of the e-STROBE framework is not limited to OpenJML. The framework can be used without OpenJML, to protect assertions by hand, or to protect other runtime assertion checkers.

The snapshot technique that we used in this framework is not only applicable to Java. A similar framework can be built for other programming environments, if the environment can be modified in such a way that copies of previous heap states are available to the assertion framework. Such a situation can be created with snapshots, but also through other techniques, such as byte code instrumentation.

7.3 Future work

During this research, we encountered several topics that would be interesting to explore, but were outside the scope of this project. We will list these topics in this chapter. Furthermore, we will list some minor improvements in Section 7.3.4, which do not need extensive research, but primarily take time to implement.

7.3.1 Using snapshots in any Java virtual machine

The e-STROBE-framework is built on top JikesRVM. We can therefore only prevent assertion interference in programs that run in this virtual machine, restricting the usability of our framework to this specific virtual machine. However, other virtual machines, like OpenJDK [5], currently do not provide the means to modify the memory model in the way that is required for both the STROBE and e-STROBE framework.

It would therefore be useful to create a snapshot framework that is not based on the object barriers in JikesRVM, but on a different technique to redirect heap accesses. Such a technique might be based on reflection or byte code instrumentation, techniques that are widely available in different virtual machines.

7.3.2 Detecting possible interference in multithreaded programs

The behaviour of multithreaded programs no longer only depends on its input, but can be influenced by the timing of a program. Timing errors are often hard to detect, and will only occur in some executions of the program. This makes it more difficult to detect these errors.

We therefore want to control the timing of a program in such a way, that the timing errors in a program can be reproduced. It would be even better to integrate such a technique with a tool like CalFuzzer [21], that analyses a program and executes it with timings that will likely cause interference in a program.

7.3.3 Supporting separation logic in OpenJML

Interference in a program can be removed entirely when separation logic [26] and fractional permissions [13] are used to specify the intended behaviour of a multithreaded program. Adding support for these techniques in OpenJML would remove interference entirely, and therefore remove the need to prevent interference during assertion checking.

7.3.4 Minor improvements

The current e-STROBE framework is still a prototype, and does not protect every heap access of a program. The biggest limitation is probably the @ConcurrentCheck annotation, that must be placed above all methods that execute code in a snapshot scope. This prevents the use of the framework to check external libraries, that will most likely not contain these annotations. The framework can be improved by activating snapshots for all methods in the program. Such an implementation must be built with caution, as our attempts to do so had a serious impact on the locks of a program.

The e-STROBE framework does not yet protect native typed fields and static fields. Implementing barriers for these fields will probably be straightforward. The biggest challenge will be to prevent recursive barrier calls. The snapshot code in the barrier uses native types for its properties, and should be rewritten to use a specialized field access, that does not trigger barriers.

The snapshot support in OpenJML is built as a proof of concept, and only applies snapshots to the checks on the pre-state and post-state of a method. Apart from the necessary modifications in OpenJML, extending this support to the entire feature set of OpenJML requires an analysis of all features, to determine which generated code should be executed in a snapshot and which code should be executed on the live state of the program.

Appendix A

Compiling and installing the e-STROBE framework

This appendix describes the installation and use of the e-STROBE framework. It contains the instructions for installation from source and for the use of a precompiled version, that has been tested on a Debian Linux (Wheezy) system.

A.1 Installing from source

The e-STROBE framework has been built using a Debian Linux (Wheezy) system with the following packages:

- openjdk7
- build-essential
- ant

The sources of the project can be obtained by sending an e-mail to the author, or downloading them from:

`http://www.dnd.utwente.nl/~jorne/e-strobe/e-strobe-source.tar.gz`

A.1.1 Compiling the framework

The files can be compiled with the *buildit* script, provided by JikesRVM. Go to the RVM folder and use the RVM build tool to compile the project:

```
bin/buildit --clear-cache -j /usr/lib/jvm/java-1.7.0-openjdk-i386/
    localhost production
```

This build script will place the compiled RVM in the /dist/production_ia32-linux/ folder. More information about this script can be found on the JikesRVM website:

`http://jikesrvm.org/Using+buildit`

A.2 Using the framework

A pre-compiled version of the e-STROBE framework is available at:

`http://www.dnd.utwente.nl/~jorne/e-strobe/e-strobe-linux.tar.gz`

We will presume that the framework is installed in the e-strobe/ folder.

RVM does not ship with a compiler. Programs must be compiled using a standard Java compiler. Add rvmrt.jar to your build path to use the STROBE features in a program:

```
javac -cp .:e-strobe/rvmrt.jar Node.java
```

The rvm command is used to run a program:

```
e-strobe/rvm Node
```

Appendix B

Compiling and using OpenJML

This appendix describes the installation and use of OpenJML in the e-STROBE framework. We provide both the changes to the OpenJML source and a compiled version of OpenJML, that can be directly used with the e-STROBE framework.

B.1 Building OpenJML with STROBE extensions

The OpenJML build process uses the Eclipse editor. [6] Build instructions for this project can be found on the OpenJML website [2].

The changes for the e-STROBE framework can be applied afterwards, using the following patch:

```
http://www.dnd.utwente.nl/~jorne/e-strobe/openjml-e-strobe.patch
```

The e-STROBE extensions require the JikesRVM runtime libraries. Therefore, the following libraries should be copied from the e-STROBE framework into the OpenJML/otherlibs/ directory

- jksvm.jar
- rvmrt.jar

OpenJML can now be built using the normal build process, described on the OpenJML website.

B.2 Running OpenJML with STROBE extension

The compiled sources for OpenJML are available in either a .zip or .tar.gz archive:

```
http://www.dnd.utwente.nl/~jorne/e-strobe/openjml-e-strobe.zip
http://www.dnd.utwente.nl/~jorne/e-strobe/openjml-e-strobe.tar.gz
```

The e-STROBE OpenJML library can be used in the same manner as the normal OpenJML library. The examples in this thesis were compiled with the following options:

```
java -jar ~/openjml-original/openjml.jar -rac -no-internalspecs *.java
```

Bibliography

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

[2] OpenJML - formal methods tool for Java and the Java Modeling Language (JML) . `http://openjml.org/`, December 2013.

[3] CFR - another java decompiler. `http://www.benf.org/other/cfr/`, August 2014.

[4] Java Thread Primitive Deprecation. `http://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html`, August 2014.

[5] OpenJDK. `http://openjdk.java.net/`, August 2014.

[6] The Eclipse Editor. `http://www.eclipse.org/`, August 2014.

[7] Edward E Aftandilian, Samuel Z Guyer, Martin Vechev, and Eran Yahav. Asynchronous assertions. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications - OOPSLA '11*, page 275, New York, New York, USA, 2011. ACM Press.

[8] 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.

[9] Wladimir Araujo, Lionel C. Briand, and Yvan Labiche. Enabling the runtime assertion checking of concurrent contracts for the Java modeling language. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 786, New York, New York, USA, 2011. ACM Press.

[10] Arthur J Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, (5):757–763, 1966.

[11] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

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

[13] John Boyland. Checking interference with fractional permissions. *Static Analysis*, 2003(9984681), 2003.

[14] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, June 2005.

[15] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31(3):25–37, May 2006.

[16] David R. Cok. OpenJML: JML for Java 7 by Extending OpenJDK. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 472–479, Berlin, Heidelberg, 2011. Springer-Verlag.

[17] David R Cok. OpenJML User Guide. 2013.

[18] David R. Cok. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. *Electronic Proceedings in Theoretical Computer Science*, 149:79–92, 2014.

[19] H.H. Goldstine and J. Von Neumann. *Planning and Coding of Problems for an Electronic Computing Instrument.* Report on the mathematical and logical aspects of an electronic computing instrument. Institute for Advanced Study, 1947.

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

[21] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 675–681, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] Gary T Leavens and Yoonsik Cheon. Design by Contract with JML, 2006.

[23] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, October 1992.

[24] Jeffrey Palm, Han Lee, Amer Diwan, and J. Eliot B. Moss. When to use a compilation service? *SIGPLAN Not.*, 37(7):194–203, June 2002.

[25] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. *ACM SIGPLAN Notices*, 2005.

[26] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[27] Edwin Rodriguez, Matthew Dwyer, Cormac Flanagan, John Hatcliff, Gary T Leavens, et al. Extending jml for modular specification and verification of multi-threaded programs. In *ECOOP 2005-Object-Oriented Programming*, pages 551–576. Springer, 2005.

[28] Alan M Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.

[29] David Anthony Watt and Deryck F Brown. *Programming language processors in Java: compilers and interpreters.* Pearson Education, 2000.

[30] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for java. In *ACM SIGPLAN Notices*, volume 40, pages 439–453. ACM, 2005.