Checking Assertions in Concurrent Programs with Active Testing

Lars Stegeman
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
l.stegeman@student.utwente.nl

ABSTRACT
This paper discusses checking assertions in concurrent Java programs. Assertions are used to specify behaviour of the program that is assumed by the programmer. The behaviour of a single threaded program is only determined by its input and all the flaws will become visible with unit testing depending on input. However most programs are concurrent or multithreaded. This means that more threads are running at the same time and not only input influences the behaviour of the program. The timing of the concurrent program also plays a vital role in the execution, however unit testing will not expose these flaws in the program. In this paper Calfuzzer is used to influence the underlying scheduler to control the timing of the program, thus exposing the timing issues in the program concerning assertions. An algorithm is proposed to explore different interleavings which can be interesting for the correctness of assertion statements. This algorithm is then tested on a number of Java programs to show its viability.

Keywords
runtime checker, assertions, active testing, concurrent programs.

1. INTRODUCTION
This introduction discusses all the main concepts used in the paper. First we discuss concurrent programs and their added complexity compared to single threaded programs. The second part will discuss the implementation of assertion statements in the Java programming language. The architecture of the Calfuzzer tool will be explained in a separate section, Section 2.

1.1 Concurrent programs
Sequential or single threaded programs are fully defined by the input values that are provided. A sequential program will execute a single stream of operations, however this paper deals with checking assertions in concurrent or multithreaded programs. Concurrent programs are much more complex because they operate with several streams of operations which may execute concurrently. Each stream executes as it would in a sequential program, but streams can interfere with one another.
Given this mechanism concurrent programs are not only defined by the input values but also the timing and order in which the threads are executed. Java does not give any guarantees about when a threads get executed, it only guarantees fairness. This makes testing concurrent programs a difficult task. This paper focusses on the correctness of the assertions which are present in the program. A single execution will not guarantee that the assertions will never fail because it can under a different timing. Therefore we automate testing these programs under different timings, to make a program more reliable.

1.2 Assertions
An assertion is a statement that lets you test assumptions about your program. Assertions are boolean expressions specifying the intended behaviour of the program, which are runtime checked. As a programmer you wish this statement to be true. However when it executes and it is false the assertion will throw an exception. The purpose of these assertions is to give your program more reliability and the programmer a tool to specify unwanted behaviour. This paper focusses specifically on the Java programming language and the syntax used in Java is: *assert(boolean expression)* [3]. Assertions are typically used in different kind of scenarios:

- The else clausule of a multiway if-statement where the programmer assumes a few statements to be true after all the above if-statements failed.
- In a switch statement in which the default case should never be executed. To achieve this a programmer can put *assert(false)* in the default case.
- For the control flow of a program. If you assume a program should never get to a certain point in the program, inserting an *assert(false)* will catch this unwanted behaviour.
- In preconditions and postconditions to check if a certain condition holds before and after the method is executed.

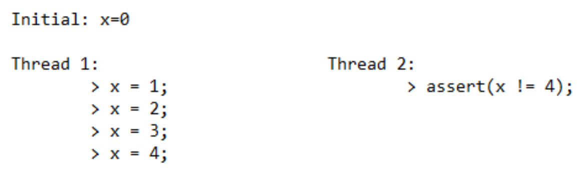
This paper focusses only on the assertions that are mentioned in the first and last bullet point. This means that only assertions which specify behaviour within the expression are used in this research. This is because the other scenarios which use an assertion to specify control flow or the default case in a switch statement need a completely different test to be able to determine whether or not they will fail.

1.3 Problem Statement
To test a concurrent program a single execution does not give any guarantees about correctness of the program.

Therefore testing concurrent programs is difficult. Programs can be made more reliable by adding assertions to the existing code, however they only give reliability in concurrent programs if they are tested over different interleavings of the program. Interleavings which give rise to an assertion error will not always occur, but can happen when executed on a different machine or with a different load. This means that controlling the scheduler explicitly is necessary when testing concurrent programs. Given this problem statement we can obtain the research question: *How can we create an environment in which we can test assertions during runtime, and perform this test on different interleavings of the concurrent program.*

1.4 Motivating Example

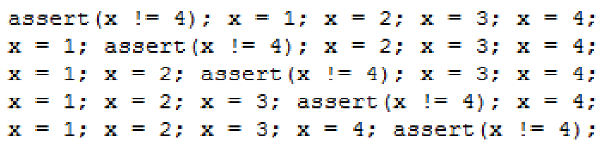
To show that this problem is relevant an example program can be seen in Figure 1. Thread 1l changes the value of x four times. Thread 2 will inspect the value x and if it is equal to four it will execute *assert(false)*. There are five different interleavings of the program, as seen in Figure 2. This small example shows that four out of five interleavings will be correct while only one interleaving will cause the error. When running the program normally it is not guaranteed that all the interleavings are executed, thus controlling the scheduler in concurrent testing is important.



```
Initial: x=0

Thread 1:                        Thread 2:
        > x = 1;                         > assert(x != 4);
        > x = 2;
        > x = 3;
        > x = 4;
```

Figure 1. Small concurrent program



```
assert(x != 4); x = 1; x = 2; x = 3; x = 4;
x = 1; assert(x != 4); x = 2; x = 3; x = 4;
x = 1; x = 2; assert(x != 4); x = 3; x = 4;
x = 1; x = 2; x = 3; assert(x != 4); x = 4;
x = 1; x = 2; x = 3; x = 4; assert(x != 4);
```

Figure 2. All possible interleavings of the small program

1.5 Roadmap

This research uses Calfuzzer as a tool to control the scheduler. Section 2 will give an overview of the architecture of Calfuzzer and how it can be extended to perform custom analysis. Section 3 will explain the choices made when designing the scheduler algorithm. Section 4 gives an overview of all the classes that are implemented to achieve the results. Section 5 presents the results of this research and the corresponding test cases. This gives insight in the usability of the algorithm and what its strengths and weaknesses are. Section 6 concludes this paper and gives possible future work.

2. CALFUZZER

To solve the problem described above we use the tool Calfuzzer. Calfuzzer describes itself as an extensible active testing framework [1]. The framework is written in Java and can effectively detect deadlocks [2], dataraces [5] and atomicity violations [4] in multithreaded programs. Calfuzzer does so by explicitly controlling the underlying scheduler of the program, it detects these errors by forcing the program into different interleavings. The technique which is used is called Active Testing, which means that data from a previous run is used in the next run to detect potential bugs. This technique uses the scheduler to force the program into a path where the bug will actually occur. Calfuzzer is based on this principle and works in two different phases. The first phase is the detection phase, in this phase the program is run once with a specific analysis. In this run all the information that is needed for the analysis is gathered. This information is used to determine where potential bugs might occur. The next phase will use the thread scheduler to verify if the potential bugs are real bugs in the concurrent program. During this phase the program that is tested will be executed several times, and Calfuzzer forces the program into different interleavings each time. These interleavings are all potential interleavings which can produce the error which is tested. If such error is produced the framework gives the corresponding execution path and displays which statement creates the exception. As mentioned above Calfuzzer already has the ability to detect deadlocks, dataraces and atomicity violations with the corresponding static and active phase implementations of the testing framework.
The next sections describe all the concepts and their implementation in the framework.

2.1 Interfaces

The framework provides two main interfaces which are used to implement custom analysis classes.

- *Analysis*: This gives access to all the functions in which users can control the scheduler. More on these functions in Section 2.5. This defines all the concurrent breakpoints that can be used to create new ActiveChecker objects.
- *ActiveChecker*: This is the active testing interface which can be implemented to create a custom scheduler. This scheduler is called each time a concurrent breakpoint is reached. The programmer has access to the different threads that are executing and paused at this point. The active checker algorithm is implemented in this class.

2.2 ANT

Calfuzzer uses ANT [6] to build and manage the framework. The build process of Calfuzzer is complex because the test cases have to be compiled separately from the framework. The framework is compiled normally using an ANT build file but the test are compiled only when they are tested, hence test classes are only build when a specific test is instantiated. A special run file is used to control the execution and order in which certain analyses are executed.

2.3 Active test data

Active Testing uses the results from previous tests in the next test. This data can not be saved in memory because the process is executed separately for each test, this means that a new virtual machine is created. Therefore the file system is used to save the data. This data is provided to each test run in the constructor and can thus be used by the current test. This is all controlled when the framework is initialized.

2.4 Monitor threads

Calfuzzer provides low priority monitor threads. The framework runs in the same virtual machine as the tested programs, this means that the complete framework can go into a deadlock or livelock. To prevent this from happening low priority monitor threads are executed and will remove one thread from the paused set when a deadlock or livelock happens. Deadlocks are detected when all the other threads are in the paused set, and livelocks happen when one thread loops without synchronizing with the other threads.

2.5 Soot

Soot[7] is used to create custom callback functions. These functions are inserted after various synchronization processes or shared memory accesses. Calfuzzer uses the Soot framework to instrument the Java bytecode instead of changing the Java Virtual Machine or Java source code. The reason for this is that the Java Source code changes more frequently and their sources may not be available for libraries. This means that the test programs that are going to be analyzed have to be compiled differently from the framework source code. This happens when a specific test program with the corresponding analysis is invoked. When the test program is run the program will execute the callback function and the designed analysis by the user is executed. With these callback functions threads are correctly paused and activated, giving the expected result. A few examples of callback functions that are used in this research are:

- *ReadBefore*: inserted before a variable is read.
- *WriteBefore*: inserted before a variable is edited.
- *MethodEnterBefore*: inserted before the method.
- *OnJoin*: inserted after two threads join.
- *NewExprBefore*: inserted when a new object is created.

3. DESIGN

In order to answer the research questions Calfuzzer will be extended in such a way that it can recognize assertions. This will be explained in Section 3.1. Next is the storage of the assertions for them to be used in the different analyses. The choices made for this are explained in Section 3.2. Calfuzzer works in two phases. The first phase is the detection phase, in which the test program will be run once. This will be used for the detection of assertions, details are in Section 3.3. The second phase is the active testing phase. The test program will be run multiple times per detected assertion, details and pseudocode of the algorithm are in Section 3.4.

3.1 Assertion detection

The tool Calfuzzer does not provide a callback function from every statement. It only provides callbacks from methods and reads and writes and various synchronization processes. To detect assertions in test programs, a small change has to be made to detect all the assertions in the program. This is easily achieved with a static method with a unique name, that will simply pass the boolean expression to a real assertion statement. These changes have to be made to all the test programs in order to make it work for the tool. These static methods are recognized by Calfuzzer as the assertion statement. As an example the statement *assert(expr)* would be transformed to *cAssertion.cAssert(expr)* in order for the tool to recognize the statement.

3.2 Designing the Active Test Object

We use active testing to test the program. This means that the program has to be run multiple times to get meaningful results, therefore an active test object has to be designed and stored on the filesystem. Before each test this object is loaded into memory and after the test is done the changed object should be written back to the file system. It is important for this object to contain all the information a test needs to complete its task. This object should contain all the information about a single assertion statement. The constructor of an AssertionObject needs its unique identifier and the line number and file in which it is called. This unique identifier is created when compiling the test program with the Soot framework. The line number of the assertion statement is later used to determine which memory locations it reads to determine the expression inside the statement. Memory accesses and writes are also assigned an unique identifier and line number can be derived from these unique identifiers through a map that is generated after the compilation. This map maps unique identifiers to its corresponding line number. This map is used to determine which memory locations are read during an assertion, and the unique identifiers of these reads and writes are stored in the AssertionObject. The object also provides some handy methods to retrieve the data easily in the correct format. The construction of the Active Test Object is completely done in the first phase of the test. During the second phase this object is used to make decisions in the ActiveChecker algorithm. These decisions are based on probabilities, and these probabilities are also stored in the object. On creation all the probabilities are one, as there is no more information available and no active test has been executed. During the active phase these probabilities will be used and changed in such a way that as many different interleavings as possible are executed.

3.3 First Phase

The first phase of the Calfuzzer framework can be used to detect assertions and save them for later use. This should be an implementation of the *Analysis* interface that is provided by Calfuzzer. Whenever an assertion is detected a new active test object should be created and stored. Also whenever a write happens to the same memory location that the assertion reads, the unique identifier of the callback should be stored as this is important for the algorithm and can be an potential error when executed with a different timing.

3.4 ActiveChecker Algorithm

The second or active phase of Calfuzzer means that the test program is run multiple times. The first phase already dealt with the detection of assertions and can be used in this algorithm. Whenever an unique identifier that is stored in the active test object is encountered, the active checker algorithm should be executed. This can either be an assertion statement or a write statement, the algorithm deals with these in the same way. The algorithm should be implemented using the provided interface *ActiveChecker* from Calfuzzer.
Pseudocode of the algorithm can be seen in Algorithm 1. In the algorithm it is assumed the AssertionObject is filled up correctly with all the breakpoints and probabilities. This means that the program is run and a breakpoint statement is reached, it will be paused given a certain probability. It can also be seen that all threads can be paused, but the framework will cope with this problem and that is why it is not dealt with in this algorithm.

Algorithm 1 ActiveChecker algorithm

```
1:  $breakpoints \leftarrow$ AssertionObject unique identifiers
2:  $probability \leftarrow$ AssertionObject linked probabilities
3:  while $Transitions[s] \neq \emptyset$ do
4:      $t \leftarrow$ next transition to be executed
5:      if $t \in breakpoints$ then
6:          if $probability[t] \geq random()$ then
7:              $probability[t] \leftarrow probability[t]/2$
8:              add $t$ to $paused$
9:          else
10:             execute $t$
11:     else
12:         execute $t$
13:     $s \leftarrow s'$   $\triangleright$ s gets the new state after execution of t
```

4. IMPLEMENTATION

The designed features are all implemented in the Calfuzzer framework and an overview of the implemented classes can be seen in Figure 3. The classes above the horizontal line are the interfaces provided by Calfuzzer, below it are the implemented *Analysis* classes which are used for this paper. The following will give a brief description of each

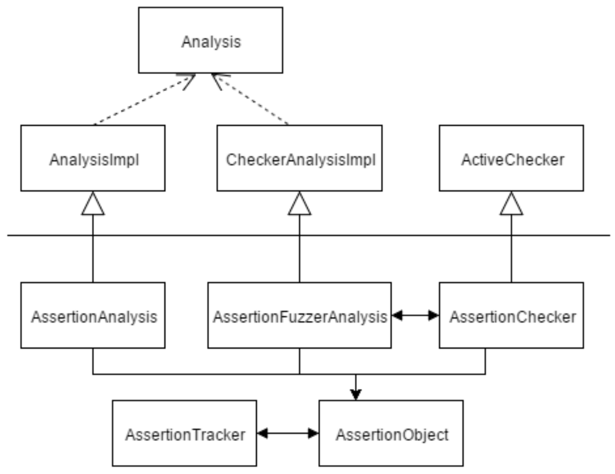


Figure 3. Overview of the implemented classes

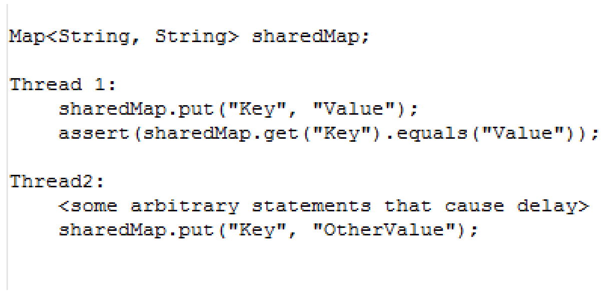
class.

- *AssertionAnalysis*: implements the first phase of the test. Detects and stores the assertions
- *AssertionFuzzerAnalysis*: implements the second phase analysis, is run multiple times focussing on one assertion at the time
- *AssertionChecker*: the custom scheduler, the ActiveChecker algorithm of Algorithm 1 is implemented in this class
- *AssertionObject*: implementation of the active test object which is used by all the analyses and algorithms
- *AssertionTracker*: keeps track of AssertionObject and gives utility functions to access the AssertionObjects

The framework is build and executed using ANT. ANT uses *build.xml* to build the whole framework and *run.xml* to execute a certain analysis. The assertion analysis is correctly added to the *run.xml* file so that it can be executed in the same way as the other analyses are executed. For the full details on the code and to build the framework yourself, the code is available at: `http://fmt.cs.utwente.nl/education/bachelor/293/`.

5. EVALUATION

This section covers the evaluation of the proposed solution. Our implementation is tested on a specific set of test programs that are created especially for this purpose. These test programs are implemented in such a way that they have the simplified behaviour of a real concurrent application. The two programs used for the evaluation are the example given in the introduction, see Figure 1, and a shared map that is accessed by multiple threads. A simplified version of the shared map can be seen in Figure 4. Both of the programs are run normally and with the



```
Map<String, String> sharedMap;

Thread 1:
    sharedMap.put("Key", "Value");
    assert(sharedMap.get("Key").equals("Value"));

Thread2:
    <some arbitrary statements that cause delay>
    sharedMap.put("Key", "OtherValue");
```

Figure 4. Example concurrent program with shared map

implemented assertion analysis. The experiments are conducted on a laptop with a 2.5 Ghz Intel i5 processor and 4GB RAM. The results of these executions can be seen in Table 1.

- The runtime is significantly longer. This is caused by the extra delay added by the algorithm.
- On a normal execution the assertion never fails, whereas the execution with the tool reports the assertion error a few times in a run with 10 trials. This happens to both of the test cases. The tool does not find the assertion error on every test run, this is because in both test cases there are more interleavings which are correct and only a few which cause the assertion to fail.

6. CONCLUSION

The research question for this paper was to create an environment in which we can test assertions at runtime, and perform this over different interleavings. This environment was created with Calfuzzer. Calfuzzer was extended with a custom analysis and its custom scheduler to control the different threads that are present. Calfuzzer was extensible and the different analyses are implemented in a direct manner. These analyses control the scheduler in such a way that interesting interleavings with the assertions are executed. The framework does not recognize assertions by default, therefore a small syntactic change has to be made to all the assertions. Test cases are tested and appear to generate different interleavings compared to a normal execution. This gives the assertions a higher level of reliability therefore making these concurrent programs more reliable.

Table 1. Results of test cases

		Assertion exception		Avg. runtime	
Program	# of executions	Norm	Tool	Norm	Tool
Shared Map	10	0	7	0.255	1.646
Introduction Example	10	0	3	0.253	0.953

6.1 Future Work

The most used assertions are pre- and postconditions. These are not recognized by the tool, therefore they have to be rewritten in order for them to be included in the algorithm. Further research could automate this process and include pre- and postconditions by default in the algorithm.
Furthermore the algorithm does not function with *assert(false)* statements as it does not contain any variables itself, however these variables are usually present in the context. An extra analysis can be added to detect these variables and include them in the Active Test Object.
Another weakness in the tool is that it will delay programs significantly when presented with a loop that changes a variable which is stored in the Active Test Object. Each iteration of the loop would cause the tool to run the algorithm and make a decision whether or not to delay the thread that is currently in that loop. This means that the algorithm will also cause a significant slowdown when presented with a larger program with more lines of code. However more research has to be done in order for this tool to be used for these larger programs.

7. REFERENCES

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

[2] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. *ACM Sigplan Notices*, 44(6):110–120, 2009.

[3] Oracle. Programming with Assertions. `http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html`. [Online; accessed 2-January-2016].

[4] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 135–145. ACM, 2008.

[5] K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Notices*, volume 43, pages 11–21. ACM, 2008.

[6] N. Serrano and I. Ciordia. ANT: Automating the Process of Building Applications. *IEEE Software*, 21(6):89–91, 2004.

[7] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

.