Model-based test case generation for source code analyzers

S.L.C. Verberkt*
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands
s.l.c.verberkt@student.utwente.nl

1. INTRODUCTION

Software engineering is the process of solving customers’ problems by the systematic development and evolution of large, high-quality software systems within cost, time, and other constraints [14]. A phase of the systematic development of software to assure that the quality and functionality meet the customers’ expectations is quality assurance. Within this phase, testing is very important. Testing is the process of systematically executing software to see if it behaves as expected.

There are two categories of testing software: black box and glass box [14, 11]. In black box testing, an engineer supplies the system with input and sees whether the output is the one that was expected. In glass box testing, the structure and the internals of the software are not considered. The engineer does not have to know about the internal functioning of the software.

Glass box testing, or white box testing, requires knowledge of the internals of the system [14, 11]. During glass box testing, the engineer does know about the code.

The present research uses black box testing. Glass box testing is not applicable because it also requires knowledge about the internals of the source code analyzer.

1.1 Test cases

During testing, the software under test is executed with a set of test cases [14, 11]. A test case is an explicit set of instructions for the system under test. The behaviour of the system for these test cases is evaluated to determine if the system is performing as expected.

In traditional software engineering, the creation of test cases is a manual process, which is highly effort-consuming and error-prone [14, 11]. In this research, we will focus on automating the creation of test cases in order to make the process of test case creation less effort-consuming and less error-prone.

1.2 Test case generation

There are two main approaches to test case generation: the static approach and the dynamic approach [1]. The static approach is to generate test cases based on a model of the test data [12]. This method is also called model-based test case generation. The dynamic approach generates test cases by executing the system repeatedly while employing criteria to rank the quality of the generated tests [21].

This project focuses on the static, model-based, test case generation. This means creating a model of the test data and using this model to systematically create test cases [1, 12]. It is possible to do this generation by following a model that specifies all options pathwise. A pathwise generator finds all possible paths based on the model. In this research, all these paths will be usable test cases.

1.3 Testing process

The testing process for source code analyzers can be divided into several phases [23, 22]. In the phase of test case generation, on which this research focuses, a suite of test cases is generated using a model of the test data.

The whole testing process for source code analyzers has some other phases, as illustrated in figure 1. Before the test case generator will be usable, one needs to build a model of the test data as input. The test case generator creates a suite of test cases as output. These test cases can be used as input for the source code analyzer that is under test. One can verify the source code analyzer by checking if the output generated by the source code analyzer is expected for the inserted test case.

This project focuses on Java source code analyzers. Therefore, the test cases will be Java source code. Since the analyzers require source code as input, the test cases can be generated completely based on the model because source code is built using a formal specified programming lan-
guage [24]. There is no need for intelligent input generation and it can be done deterministically.

1.3.1 The model of the test data
The test case generator requires a model of the test data to be able to create test cases. In this project, the model is based on a template which is applicable to all test cases in the suite. In this template, there are several points that differ between the various test cases. We will call these points variation points. By finding all possible options for the several variation points and combining this with the template, one can build a model of the test data.

1.3.2 Source code analyzers
Source code analyzers can be divided into two types: static and dynamic [15]. Static source code analysis is performed without actual execution of the source code [3]. A static analyzer takes the source code for input and performs the task it was designed for. Dynamic analysis, on the other hand, does execute the code. In this research we will focus on both types, since they both require source code as test cases.

1.4 Proof of concept implementation
The goal of this research is to build a model-based test case generator for source code analyzers. We will first focus on a specific implementation for testing purposes. This specific proof of concept will be the generation of test cases for analyzers that discover data races based on lock-based synchronization.

After implementing a working, but not full-fledged, specific test case generator, we will separate the specific parts and generic parts of the implementation. When these parts are separated, the implementation will be a modular test case generator, which can be adopted for any source code analyzer.

2. PROBLEM STATEMENT
Manual test case creation is an extremely effort-consuming and error-prone process [18, 11]. According to Philips et al. [18], creating a model-based test case generator is possible and most likely an improvement over manual test case creation. This test case generator decreases the manual effort required to build a good testing environment. It is also less likely that some part will be forgotten in the testing process.

The goal of this research is to create a model-based test case generator for source code analyzers using Java technology. This test case generator must be able to generate a suite of test cases featuring all possible variations of the supplied model of the test data. The model of the test data is based on the variation points among the desired test cases. A variation point is a specific point that differs between test cases.

The deliverable outcome of this research is the test case generator. It has a modular design to be able to adopt it for the testing of other potential errors. This test case generator reduces manual labour while testing and facilitate common testing techniques such as unit testing.

2.1 Research questions
Following from the problem statement, the main question of this paper is:

- Is it possible to create a model-based test case generator for source code analyzers for Java source code?

In order to answer this question, a non-generic test case generator to test analyzers that discover data races based on lock-based synchronization is the subject of research first. Therefore, the research starts with detecting the possibilities of a specific test case generator. This specific test case generator is the basis for the generic implementation. Further research questions are:

1. What are the variation points of a test case for an analyzer that discovers data races based on lock-based synchronization?
2. How can we generate multiple test cases based on the found variation points?
   a. Which technology can be used for this generation?
   b. Can this generator be made configurable?
3. Is it possible to generalize the implementation to a model-based test case generator for source code analyzers?

To answer these questions, a test case generator has been built as deliverable outcome. The generator has a modular structure, which can be adopted to other source code analyzers.

3. RESEARCH APPROACH
The model-based test case generator delivered by this research consists of a generic and a specific part. This architecture is discussed in subsection 3.1. To generate test cases, the generator requires a model of the test cases. The approach to build this model is discussed in subsection 3.2. When these two parts are combined, a complete model-based test case can be delivered, which is discussed in subsection 3.3.

3.1 Architecture of the test case generator
The built test case generator consists of a set of general Java classes and a set of specific Java classes. The generic classes are reusable for other test case generators and the specific classes implement the generation of a set of test cases for an analyzer that discovers data races based on lock-based synchronization. The generator uses an external templating library, which is called StringTemplate [17].
The architecture of the system is illustrated in figure 2. The system has a Main class, whose sole purpose is to execute the TemplateLoader. The TemplateLoader has the purpose to load a Template and generate test cases for all possible combinations of variation points. These are saved by using the FileOutput class. The StringTemplate library is used as a templating system to create the code for the test cases. The Template of the test case is saved in a separate StringTemplateGroup file, which is used in the Template class. Both, classes Template and TemplateLoader, need a specific implementation. In this specialization, the model of the test cases will be implemented.

### 3.2 Model of a test case

The model of a test case consist of three parts: a template, a set of binary variation points and a set of conditions for the variation points. A variation point is a specific point, which differs between the generated test cases. All variation points are binary, which means they are either on or off. An example of a set of variation points is listed in table 1.

<table>
<thead>
<tr>
<th>Number</th>
<th>Variation point</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP1</td>
<td>StaticField</td>
</tr>
<tr>
<td>VP2</td>
<td>SynchronizedBlock</td>
</tr>
<tr>
<td>VP3</td>
<td>StaticSynchronizedBlock</td>
</tr>
<tr>
<td>VP4</td>
<td>StaticInitializer</td>
</tr>
</tbody>
</table>

Not all combinations of variation points are valid. Therefore, there are constraints for every variation point. A constraint states which values other variation points should have for the variation point to be on. If the constraints for a variation point are not satisfied, the variation point cannot be on. An example of these constraints is shown in table 2. Using these conditions, the combinations stated in table 3 are possible.

The variation points are used to configure the template. Since all variation points are binary, they are used as on/off switches on the template. The template itself is built using StringTemplate syntax. An example template is listed in listing 1.

Listing 1. Example template for a test case.

```java
testcase (vp1, vp2, vp3, vp4) ::= <<
public class TestCase {
    public static <endif>
        int field = 2;
    <if(vp4)>
        static {
            field = field * 2;
        }
    <endif>
    public void testCase() {
        <if(vp2)>
            synchronized(
                <if(vp3)>self
                <else>this
                <endif>) {
                <endif>
                    field = field * 2;
            }
        <if(vp2)>
    }
<endif>
>>
```
Table 2. Example constraints for a set of variation points.

<table>
<thead>
<tr>
<th>Variation point</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynchronizedBlock</td>
<td>StaticInitializer is OFF</td>
</tr>
<tr>
<td>StaticSynchronizedBlock</td>
<td>SynchronizedBlock is ON and StaticField is ON</td>
</tr>
<tr>
<td>StaticInitializer</td>
<td>SynchronizedBlock is OFF</td>
</tr>
</tbody>
</table>

Table 3. Valid combinations of variation points in the example.

<table>
<thead>
<tr>
<th>VP1</th>
<th>VP2</th>
<th>VP3</th>
<th>VP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>OFF</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>ON</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
</tr>
<tr>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
</tr>
</tbody>
</table>

3.3 Test case generation

When executed, the system first constructs a specialization of TemplateLoader, which loads the the set of variation points, the template and the constraints for the set of variation points. When done with loading, the generator constructs the Cartesian product of variation points and removes all combinations that are illegal according to the constraints. For every combination, the system constructs an instance of the specialized Template, configures this using the variation points and generates a test case. To give an example, listing 2 shows a generated test case with the variation points one through three set (StaticField, SynchronizedBlock and StaticSynchronizedBlock).

Listing 2. Example of a test case based on the variation points in table 1, the constraints in table 2 and the template in listing 1.

```java
public class TestCase {
    public static int field = 2;
    public void testCase() {
        synchronized (self) {
            field = field * 2;
        }
    }
}
```

4. RESULTS

For the specialized test case generator, a model has been built. This model should cover all test cases of interest for the proof of concept generator. In the Java Language Specification [8], one can find all constructs which influence the synchronization between multiple threads and all types of fields that could be accessed in different threads in a Java application. Besides the Java Languages Specification, the model is also based on several books and articles on concurrent programming and common faults in concurrent programming [13, 7, 19, 16]. The found variation points are listed in table 4, the constraints for these variation points in table 5 and the template in listing 4. As an example, listing 3 shows a generated test case with the variation points one through five set (StaticField, VolatileField, Method, StaticMethod and SynchronizedMethod).

The test case generator generated 40 test cases using the model of the proof of concept. All these models were successfully compiled using the Java compiler. They were also used as input for the Java verification tools FindBugs and JLint. Both tools were able to handle the test cases. During testing, there where models which generated up to 1024 correct test cases.

Using the technique of this research, one reduces errors which could occur when manually creating test cases. The test case generator generates all valid variations in the model. It will not forget any test cases, and will not create invalid test cases.

For small amounts of variation points, the time to set up the model may be larger than the time to create the test cases by hand. However, for larger amounts of variation points, the test case generator reduces effort.

5. RELATED WORK

Both black box and glass box testing have been subject of research for decades. Fröhlich and Link [6] for example already wrote about black box testing. They designed a method to create test cases from use cases using a generated state model and an artificial intelligence planning language.

Philipps et al. [18] describe a way of generating test cases from a model of the command and response sequences of a smart-card. This paper concludes that model-based testing is a viable alternative to handwritten test cases, but it requires start-up work like constructing the models.

Both Jalote [11] and Lethbridge [14] explain the process of software engineering. This includes very detailed work on the manual process of test case creation which is a very good basis for the present research.

Hartman [9] created a summary of available model-based test case generation tools. These tools all have in common that they use a model as an input and a set of generation directives to create the test cases.

Tretmans and Brinksma [23, 22] wrote about the tool TorX, a tool for formal testing. The TorX tool also automates the complete testing process and does on-the-fly testing. On-the-fly means that test derivation and test execution are performed simultaneously as opposed to batch-wise generation where all test cases are generated before any test execution occurs [4]. An important difference between TorX and this research is the model. In this research, test cases are generated based on a model of the test data, in TorX they are generated based on a model of the system.

Beyer et al. [5] and Henzinger et al. [10] researched a test case generation system with a model checking approach. This tool, BLAST, checks temporal safety properties of C programs.

Microsoft Research [20] created a glass box test generation tool called Pex. Pex automatically produces a small test suite with high code coverage for a .NET program. Because of the glass box implementation, Pex is not of use for the present research. The deliverable outcome of this research will be usable for more Java source code analyzers, which is not possible when the test cases are generated based on the internals of one source code analyzer.

There has also been research on a complete testing framework by Artho [1, 2]. In this framework, test case generation is combined with runtime verification, which creates a fully automatic testing process. This gives a basis for embedding this research in a framework.
6. CONCLUSIONS

In this section, the research questions will be discussed in order to answer the main question of this research, Is it possible to create a model-based test case generator for source code analyzers for Java source code?

The variation points mentioned in the first research question, What are the variation points of a test case for a analyzer that discovers data races based on lock-based synchronization?, are listed in table 4. The variation points in this table are binary, so they can either be on or off. There are also conditions for several of the listed variation points found, which should be satisfied for the variation point to be true. These conditions are listed in table 5.

The answer to the second research question, How can we generate multiple test cases based on the found variation points?, depends on the architecture of the system. The built test case generator uses templating technology combined with the model of the test cases. The test case generator is also configurable. One could edit the variation points, the template, or the conditions for the combinations of variation points.

The last research question, Is it possible to generalize the implementation to a model-based test case generator for source code analyzers?, can also be answered positively. The architecture for the test case generator consists of a generic and a specific part. Therefore, one only needs to build a model of the test cases to be generated in order to implement the test case generator for another sort of source code analyzer.

Therefore, the answer to the main research question is: Yes, using the software delivered by this research, one could create a model-based test case generator for source code analyzers for Java source code. The built generator reduces effort and errors in the process of creating test cases.

7. FURTHER WORK

Further work based on this research could try to predict the outcomes of the several test cases. With this research, one could build a test suite and test if an analyzer is compliant with the generated code. In further research, it could be an objective to also predict what the analyzer should say about the test case.

8. ACKNOWLEDGMENTS

I would like to acknowledge:

- dr. ing. C.M. Bockisch, the technical supervisor of this project.
- dr. H. Sozer, the track supervisor on behalf of the software engineering group.
- dr. M.I.A. Stoelinga, the track supervisor on behalf of the formal methods and tools group.

9. REFERENCES


Listing 3. Template for a test case for an analyzer that discovers data races based on lock-based synchronization.

```java
main() ::= <<
public class Main {
    public static final FirstThread firstThread = new FirstThread();
    public static final SecondThread secondThread = new SecondThread();

    public static main(String[] args) {
        (new Thread(firstThread)).start(); (new Thread(secondThread)).start();
    }
}
>>
field_class(field, static, initializer, method, constructor) ::= <<
public class Field {
    public static Field instance;
    <field> = 2;

    public Field() {<constructor> }
    <static_initializer>
    <method>
    public static synchronized Field getInstance() {
        if (instance == null) { instance = new Field(); } return instance;
    }
}
>>
first_thread(method_call, synchronized_block) ::= <<
public class FirstThread implements Runnable {
    public void run() {
        <synchronized_block>
        <method_call>
    }
}
>>
second_thread(method_call, synchronized_block) ::= <<
public class SecondThread implements Runnable {
    private Field reference;
    public void setReference(Field ref) { reference = ref; }

    public void run() {
        <synchronized_block>
        <method_call>
    }
}
>>
field(vp1, vp2) ::= <<
public <if(vp1)>static <endif><if(vp2)>volatile <endif>int field
>>
field_call(vp1) ::= <<
<if(vp1)>Field<else>Field.getInstance()<endif>.field
>>
method(vp3, vp5, vp4, field_call) ::= <<
<if(vp3)>
    public <if(vp4)>static <endif><if(vp5)>synchronized <endif>void method() {
        <field_call> = <field_call> * 2;
    }
<endif>
>>
method_call(vp3, vp4) ::= <<
<if(vp3)>Field<else>Field.getInstance()<endif>.method();<endif>
>>
synchronized_block(vp6, vp7, field_call) ::= <<
<if(vp6)>synchronized(<if(vp7)>self<else>this<endif>) {
            <field_call> = <field_call> * 2;
        }
<endif>
>>
constructor(vp8, vp9, field_call) ::= <<
<if(vp8)>
    Main.secondThread.setReference(<if>(this));
    <endif>
    <field_call> = <field_call> * 2;
<endif>
>>
static_initializer(vp10, field_call) ::= <<
<if(vp10)>static {<field_call> = <field_call> * 2; }
<endif>
>>
```
Table 4. Variation points of a test case for an analyzer that discovers data races based on lock-based synchronization.

<table>
<thead>
<tr>
<th>Number</th>
<th>Variation point</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP1</td>
<td>StaticField</td>
<td>Whether the field is static</td>
</tr>
<tr>
<td>VP2</td>
<td>VolatileField</td>
<td>Whether the field is volatile</td>
</tr>
<tr>
<td>VP3</td>
<td>Method</td>
<td>Whether there is a method</td>
</tr>
<tr>
<td>VP4</td>
<td>StaticMethod</td>
<td>Whether the method is static</td>
</tr>
<tr>
<td>VP5</td>
<td>SynchronizedMethod</td>
<td>Whether the method is synchronized</td>
</tr>
<tr>
<td>VP6</td>
<td>SynchronizedBlock</td>
<td>Whether there is a synchronized block</td>
</tr>
<tr>
<td>VP7</td>
<td>StaticSynchronizedBlock</td>
<td>Whether the synchronized block is static</td>
</tr>
<tr>
<td>VP8</td>
<td>Constructor</td>
<td>Whether there is a constructor</td>
</tr>
<tr>
<td>VP9</td>
<td>ReferencePublished</td>
<td>Whether the reference is published in the constructor</td>
</tr>
<tr>
<td>VP10</td>
<td>StaticInitializer</td>
<td>Whether there is a static initializer</td>
</tr>
</tbody>
</table>

Table 5. Constraints for variation points of a test case for an analyzer that discovers data races based on lock-based synchronization.

<table>
<thead>
<tr>
<th>Variation point</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>(SynchronizedBlock is OFF) and (Constructor is OFF) and (StaticInitializer is OFF)</td>
</tr>
<tr>
<td>StaticMethod</td>
<td>(Method is ON)</td>
</tr>
<tr>
<td>SynchronizedMethod</td>
<td>(Method is ON)</td>
</tr>
<tr>
<td>SynchronizedBlock</td>
<td>(Method is OFF) and (Constructor is OFF) and (StaticInitializer is OFF)</td>
</tr>
<tr>
<td>StaticSynchronizedBlock</td>
<td>(SynchronizedBlock is ON)</td>
</tr>
<tr>
<td>Constructor</td>
<td>(Method is OFF) and (SynchronizedBlock is OFF) and (StaticInitializer is OFF)</td>
</tr>
<tr>
<td>ReferencePublished</td>
<td>(Constructor is ON)</td>
</tr>
<tr>
<td>StaticInitializer</td>
<td>(Method is OFF) and (SynchronizedBlock is OFF) and (Constructor is OFF)</td>
</tr>
</tbody>
</table>

Listing 4. Test case for an analyzer that discovers data races based on lock-based synchronization.

```java
public class Main {
    public static final FirstThread firstThread = new FirstThread();
    public static final SecondThread secondThread = new SecondThread();

    public static void main(String[] args) {
        (new Thread(firstThread)).start();
        (new Thread(secondThread)).start();
    }
}

public class Field {
    public static final Field instance;
    public static volatile int field = 2;

    public static synchronized void method() {
        Field.field = Field.field * 2;
    }

    public static synchronized Field getInstance() {
        if (instance == null) {
            instance = new Field();
        }
        return instance;
    }
}

public class FirstThread implements Runnable {
    public void run() {
        Field.method();
    }
}

public class SecondThread implements Runnable {
    private Field reference;

    public void run() {
        Field.method();
    }

    public void setReference(Field ref) {
        reference = ref;
    }
}
```