Java Test Driver Generation from Object-Oriented Interaction Traces

Frank S. de Boer\(^2\) and Marcello M. Bonsangue\(^1\) and Andreas Grüner\(^3\) and Martin Steffen\(^4\)

\(^1\) LIACS, Leiden, The Netherlands
\(^2\) CWI Amsterdam, The Netherlands
\(^3\) Christian-Albrechts-University Kiel, Germany
\(^4\) University of Oslo, Norway

Abstract. Whereas object-orientation is established as one major paradigm for software development, testing methods specifically targeted towards object-oriented, class-based languages are less common. We propose a testing framework for Java programs, based on the observable trace semantics of class components, i.e., for black-box testing. In particular, we propose a test specification language which allows to describe the behavior of the test harness in terms of the expected interaction traces between the program and the tester. The language is tailor-made for Java, e.g., in that it reflects the nested calls and return structure of thread-based interaction at the interface. From a given trace specification, a testing environment is synthesized such that component and environment represent a closed program.

The design of the specification language is a careful balance between two goals: using programming constructs in Java-like notation helps the programmer to specify the interaction without having to learn a completely new specification notation. On the other hand, additional expressions in the specification language allow to specify the desired trace behavior in a concise, abstract way, hiding the intricacies of the required synchronization code at the lower-level programming language.

Keywords: black-box testing, Java, trace-based observable behavior, test-driver generation.

1 Introduction

Testing is of prime importance in assuring the quality of software. In contrast to exhaustive methods for system verification and validation, testing aims at detecting faults, thereby increasing confidence in the system under test \(^13\). To manage the complexity of modern software, it is generally accepted that testing should be systematic and integrated into the software development process. Test scripts should be generated automatically from the specification and tools should take care of the automation of the
several aspects and levels (e.g., unit, integration or system testing) of the testing framework \[4\].

The automatic generation of test cases based on state exploration can be very costly, and sometimes even impossible when only partial knowledge of the implementation is available. This is typically the case for object-oriented systems, where data and methods are encapsulated into class instances, so that sometimes operations must be added to a class interface just to support testing. An alternative to state exploration methods are scenario tests, i.e., specifications of how the system under test should behave in a specific environment \[13\]. Typically, test scenarios are written in a (formal) specification language (e.g., as message sequence charts), from which test cases can be generated automatically. For object-oriented systems, however, it may be hard to generate a valid environment implementing the test cases because the objects under test might be embedded into a complex system architecture. The generation of a simulation environment from a test case is thus often preferred to enable interactions between the system under test and the tester \[9,10,1\]. The consequence is a loss of control, possibly resulting in a simulation not reflecting the real behavior of the system as deployed, especially when multi-threading and distribution comes into play.

In this paper we define a formal specification language for test cases of \textit{Java} components. From a specification we generate the code of a \textit{Java} program which serves as the test driver of the component under test (CUT). The generated code can be deployed together with that of the CUT so to be executed in a specific architecture, without the necessity of simulating it.

We consider a \textit{Java} component to be a set of classes and their instances. Components may only communicate via method calls. In particular, fields can only be accessed by objects of the same component. This restricted access is enforced by the use of the \textit{Java} access modifier. To avoid encapsulation problems, inheritance is not allowed to cross the border between \textit{Java} components. By contrast, classes can be instantiated even by objects residing in a different component. This way, a component does not expose its implementation details and its semantics can be compositionally described by means of sequences of method calls and method returns between the component and its environment \[2\]. These sequences, also called traces, form the formal basis of our test specification language. Thus, a specification basically describes a sequence of stimuli (i.e., outgoing calls and returns), to be realized by the testing environment, and of the expected behavior (i.e., incoming calls and returns) of the CUT.
The syntax of our specification language is inspired by that of Java. It allows for the specification of sequences of incoming and outgoing calls and returns by means of usual control flow operation like while-loops and conditionals. Further it contains constructs for data manipulation, expression evaluation, and switch on incoming method calls.

The specification language and the code generation are illustrated by a voting system example. The voting system is a Java component that when activated by an initiator collects a vote from a group of external voter objects, compile a report and return it to the initiator. It can be used, for example, to detect termination of a group of objects. The specification of the example describes the scenarios to be tested in terms of the expected interactions of the CUT with the initiator object and with a set of voter objects. From this specification we generate a set of classes representing the tester-environment. Only if the CUT can execute the expected interactions, it passes the test. A failure message is returned if the CUT is not able to follow the sequence of messages as specified. By analyzing the failure message, we can decide if the failure is due to an unrealizable specification (like communication between two objects that cannot possibly know each other), a bug (like a memory fault) or because the component is fault.

The paper is structured as follows. Section 2 contains the definition of the specification language, used to describe the trace-based behavior of the observer. Section 3, the core of the paper, describes the code generation from the abstract specification. In Section 4 we discuss briefly the extension of the specification language to the multi-threaded case. Finally, Section 5 concludes with discussing related and future work.

2 The test specification language for single threaded Java

This section formalizes the specification language to describe the traces of the observer. We start in Section 2.1 with the abstract syntax of the specification language; Section 2.2 deals with failure reports.

2.1 Syntax

The abstract syntax of the test specification language is given in Table 1. In the syntax we abstract from failure reports, which are discussed in the next section. The main aspects of the syntax are explained in terms of Listing 1.1, which specifies the external observable behavior of a voting component.
Listing 1.1. Specification of the voting example

```java
import java.util.HashSet

provided
Voter {
  vote() : bool;
  fVote : bool;
}

required
Census {
  census(HashSet) : bool;
}

c : Census;
called : HashSet = new HashSet();
voters : HashSet = input.read;
conj : bool = true;

new !Census()! {
  c := return();
  !census(voters.clone()) {
    while (called.size() < voters.size()) do {
      (this : Voter)?vote() where (called.contains(this) = false) {
        called.add(this);
        conj := conj & this.fVote;
        !return(this.fVote)!
      }
    }!
  }
  x := ?return(y : bool) where (y = conj)
}
```
In general, a test specification \textit{spec} starts with a list of imported classes. These classes may only be used for specifying or analyzing the test, but must not contribute to the communication with the CUT. In our example, we import Java’s \texttt{HashSet} to store the set of voters. However, the list of imported classes is optional. It is followed by a list \textbf{provided} of class signatures designating the classes belonging to the environment. In our case, the signature of the environment is specified by the class \texttt{Voter} which contains a boolean field \texttt{fVote} and its corresponding method \texttt{vote} to get its value. On the other hand, \textbf{required} denotes a non-empty list of class signatures designating the CUT. As explained in the introduction the fields of the \textbf{provided}-classes cannot be accessed by the CUT and the \textbf{required}-classes only specify method signatures. Finally, \textit{stmt} describes the behavior of the tester and the expected behavior of the component. In our example, it first declares a local variable \texttt{c} and initializes the local variables \texttt{called}, \texttt{voters}, and \texttt{conj}. The local variable \texttt{called} is simply initialized to the empty set and the set \texttt{voters} of instances of \textbf{required}-class \texttt{Voter} is obtained by reading from a given predefined input file.

After the initialization, the actual interaction is described. First, a new instance of the component class \texttt{Census} is created by calling its constructor method and waiting for the return value which is assigned to the local variable \texttt{c}. In the specification language, we view the instantiation of a component class as a particular case of a call of a method of the component by the tester as explained in detail below. Next, the tester calls the method \texttt{census} of component object \texttt{c}. It passes a copy of the set of voters to the component (note, that passing on the set of voters itself would give rise to a shared data structure which may influence the outcome of the test). This method call resembles a method call in Java, apart from the additional exclamation mark and the missing assignment of the result to a variable. The expected behavior of the CUT is that it returns the conjunction of the votes of all voters. In order to calculate the conjunction, the component has to find out the votes of the voter objects of set \texttt{voters}. This is modeled by a call of a method of a voter by the component. Note that this method belongs to the tester. Such calls we indicate by question marks. After having received all this calls \texttt{census} returns to the tester the final result.

More specifically, \textit{after} the call of the component’s method \texttt{census} and \textit{before} the corresponding return with the final result, the component must call method \texttt{vote} of each \texttt{Voter} object which belong to the tester. Only then the invocation of \texttt{census} can return. To express this we split a call of a method of the CUT by the tester into two distinguished events, i.e., the
outgoing call itself, indicated by an exclamation mark, and the incoming return, indicated by a question mark. Dually, a call of a method of the tester by the component is described by the incoming call itself, indicated by a question mark, and the outgoing return, indicated by an exclamation mark. Both for an incoming call and an incoming return we use formal parameters to denote the actual parameters provided by the component. Values which are passed to the tester by an incoming communication can be checked by a where-expression which must evaluate to true for the actual values, otherwise the test is not successful.

In the syntax, a call event and the corresponding return event mark the beginning and end of a block construct. These different block constructs are denoted in the syntax by $s_{in}$ and $s_{out}$. In a single-threaded setting, the flow of control between the component and the tester is reflected by alternating nested block statements $s_{in}$ and $s_{out}$. These block statements form the basic building block of the language. Of particular interest is the use of $s_{in}$ statements in the context of a callswitch: which allows the specification of non-deterministic choice between incoming calls.

Apart from the interactions between the component and the tester, the specification in general also will involve internal computations. For example, for computing values for communication and driving the test execution. In the syntax these statements are denoted by $s_{int}$. Note that such an internal computation assumes that the tester has control. Thus, we cannot specify internal computations right after an outgoing call. The above example shows, however, that in practice it is convenient for driving the test execution to allow, for instance, evaluation of guards right after outgoing communication. In Section 3 we explain how to generate correct Java code for such specifications.

### 2.2 Failure reports

A terminating execution of a test specification is successful if it does not generate a failure report. In general, failures arise because of invalid assert statements and where-clauses and unspecified incoming method calls and returns. As an example of the latter kind of failures consider the following specification fragment.

\begin{verbatim}
new !C() (this : T)?m() ... x := ?return()
\end{verbatim}

This example can give rise to two kinds of failures. First, the constructor of component class $C$ returns without calling any method of the

---

[^1]: We consider here a call of a constructor as a special method call with object creation.
spec ::= [import clintf*] provided clintf* required clintf* stmt
clintf ::= c{meth* field*}
meth ::= [static] m(T, . . . , T) : T
field ::= f : T
stmt ::= s_out | s_in
s_out ::= x : T | x := e | e.f := e | assert(e_bool) | x.m(v, . . . , v)

s_in ::= skip | s_in | c_in | callswitch; c_in
| if (e_bool) then s_in else s_in | while (e_bool) do s_in

s_out ::= x : T | x := e | e.f := e | assert(e_bool) | x.m(v, . . . , v)

Note that it is possible to specify tests which always fail because of
interaction scenarios which are unrealizable. For example consider the
specification

\( x := \text{new } T(); \text{new } C() \{ (\text{this} : T)?m(u : T) \text{ where } (u = x) \ldots \} \),

which after creating an instance of a test class \( T \) calls the constructor
of component class \( C \) and then expects an incoming call with the newly
created instance of class \( T \) as actual parameter. Since the instantiation
of a tester class by the tester itself is an internal action this newly cre-
ated object is not known by the component unless the tester explicitly
communicates it.

Of particular interest are therefore static analysis techniques which
identify such failures. Since it is not possible to identify all such failures,
statically, we provide a runtime check by recording all identities of tester
objects which are known to the component. If a test execution leads to an
expectation of an incoming communication with a tester object as actual
parameter, resp., as return value which actually cannot be known by the
component, then the tester reports an invalid test execution and aborts.

Table 1. Abstract syntax
3 Code generation

In this section we describe how to generate Java code from a test specification for the methods of the tester classes. In general, to generate Java code, we first transform the test specification such that its internal computations comply with the single-threaded flow of control. Applying this transformation to our example test specification results in the specification which is given in Listing 1.2.

First, we introduce labels to mark incoming calls and returns. These labels are also used to rewrite a while statement in an equivalent statement using conditions and virtual goto statements. Such a virtual goto statement is of the form next := ℓ where next is a new auxiliary variable for the tester and ℓ is a label. The auxiliary variable next describes the control of the tester and refers to the next expected incoming call or return. The result of this transformation is that all internal computations fall within the scope of the control of the tester.

We now explain the code of Listing 1.3 generated from the above specification.

The class Tester introduces an enumeration type for the labels and it initializes the instance variables next, voters, called, and conj and declares the instance variable c. Since according to the test specification the tester starts the execution, a main method is introduced to describe its behavior. This behavior consists of first calling the constructor method after which it is checked whether next has not progressed. Note that this indicates that the constructor method itself didn’t generate calls of methods of the tester. A similar test is generated by the subsequent call of the method census. On the other hand, the implementation of the method voter consists of checking whether it has been called when expected. This expectation is expressed by a test involving the corresponding label.

In the voting example, the method vote is only called once for every Voter object. However, in general a method could be called several times in different situations where also different reactions might be wanted. We explain the general case in terms of the following two fragments of a preprocessed test specification. The overall structure of the fragment given in Listing 1.4 depicts the n occurrences of incoming calls of the method m. Furthermore, the ith incoming call contains at top level an outgoing call to m′. Finally, this latter outgoing call contains an incoming call to a method m″. This leads to the Java code fragment of method m which is shown in Listing 1.5.
import java.util.HashSet

providing Voter {
  vote() : bool;
  fVote : bool;
}

required Census {
  census(HashSet) : bool;
}

c : Census;
called : HashSet = new HashSet();
voters : HashSet = input.read;
conj : bool = true;

new !Census() {
  [ℓ₀]c:=?return();
}

if (called.size() < voters.size()) then
  next := ℓ₁
else
  next := ℓ₂;

c!census(voters.clone())!
while (called.size() < voters.size()) do {
  [ℓ₁](this : Voter)?vote() where (called.contains(this) = false) {
    called.add(this);
    conj := conj & this.fVote;
    if (called.size() < voters.size()) then
      next := ℓ₁
    else
      next := ℓ₂;
  }!
  return(this.fVote)!!
}

[ℓ₂] x:=?return(y : bool) where (y = conj)
Listing 1.3. Voting example: Java code

```java
import java.util.HashSet

class Tester {
    enum Label = {ℓ₀, ℓ₁, ℓ₂};
    Label next = ℓ₀;
    HashSet called = new HashSet();
    HashSet voters = new input.read;
    bool conj = true;
    Census c;

    void main() {
        c = new Census();
        assert(next == ℓ₀);
        if (called.size() < voters.size())
            Tester.next = ℓ₁;
        else
            Tester.next = ℓ₂;
        y = c.census(voters);
        assert(next == ℓℓ₂);
        assert(y == conj);
        x = y;
    }
}

class Voter {
    bool vote() {
        switch(Tester.next) {
            case ℓ₁:
                assert(called.contains(this) == false);
                called.add(this); 
                conj=conj & this.fVote;
                if (called.size() < voters.size())
                    Tester.next = ℓ₁;
            else
            Tester.next = ℓ₂;
        return(this.fVote);
        }
    case default:
         assert(false);
    }
```
Listing 1.4. Specification fragment: the general case

\[
[\ell_1] \text{(this : } T)\text{?m(...)} \ldots
\]

\[
[\ell_i] \text{(this : } T)\text{?m(...)} \{
\ldots
x!m'() \{
\ldots
[\ell'] \text{(this : } T')\text{?m''()} \ldots
\ldots
[\ell'] \text{return } \ldots
\}\}
\ldots
[\ell_n] \text{(this : } T)\text{?m(...)} \ldots
\]

Listing 1.5. Java code fragment: the general case

\[
T \text{.m(...)} \{
\text{switch (Tester.next) } \{
\text{case } \ell_1: \ldots
\ldots
\text{case } \ell_i: \{
\text{next = } \ell_i;
\text{x.m'}();
\text{assert(next == } \ell');
\text{next = } \ldots
\text{return } e
\}
\ldots
\text{case } \ell_n: \ldots
\}
\]
\]
Listing 1.6. specification fragment: formal parameters

\[ [\ell] (\text{this} : T)?m(x : T') \{ \\
\quad ; \\
\quad [\ell'] (\text{this} : T)?m(y : T') \{ \\
\quad \text{if } (x < y) \text{ then} \\
\quad \quad \text{this}._m'(x) \ldots \\
\quad \text{else} \\
\quad \quad \text{this}._m'(y) \ldots \\
\quad \} \}
\]

The method body consists of a Java `switch` statement which checks for the occurrence of the incoming call. The code for each such case is generated from the corresponding occurrence. For example, the outgoing call `x!m'` is preceded with the update of `next` with label `\ell` of the next expected incoming call of method `m''`. After the outgoing call we check whether `next` refers to the label of its return. Right before we return from this call of method `m` `next` is updated to the next expected occurrence of an incoming call or return.

It is also important to know that the formal parameters of incoming method calls in the specification language are different from the formal parameters of Java method definitions and therefore cannot be directly translated. To understand this, let us consider the following fragment of another preprocessed specification presented in Listing 1.6.

Here, we have two nested incoming calls of the same method `m`. However, the outer method call uses `x` as its formal parameter whereas the inner method call uses `y` and also has access to `x`. Therefore, we model these formal parameters as static variables of the `Tester` class which are globally accessible. In order to describe the scope of the variables we annotate them with the label of their occurrence. For the above specification we have then the implementation fragment given in Listing 1.7.

4 Generalizing to the multi-threaded case

We present an extension of the specification language for testing components in a multi-threaded setting. First we extend the communications between the component and the tester with an additional parameter denoting the executing thread. Second, we have to relax the nested block
Listing 1.7. Java code fragment: formal parameters

\[
T'' \ m(T' n) \{ \\
\text{switch(Tester.next) \{} \\
\text{case } \ell_1: \text{Tester.}*_x = n; \ldots \\
\text{case } \ell_2: \{ \\
\text{Tester.}*_y = n; \\
\text{if(Tester.}*_x < \text{Tester.}*_y) \\
\text{this.}m'('\text{Tester.}*_x); \ldots \\
\text{else} \\
\text{this.}m'('\text{Tester.}*_y); \ldots \\
\} \\
\}
\]

spec ::= \[\text{import clintf}^*\] provided clintf\(^+\) required clintf\(^+\) stmt

<table>
<thead>
<tr>
<th>spec</th>
<th>class intf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>clintf ::= c : c {meth* field*}</td>
<td>field decl.</td>
</tr>
<tr>
<td>meth ::= static (m(T, \ldots, T) : T)</td>
<td>meth. sign.</td>
</tr>
<tr>
<td>field ::= static (f : T)</td>
<td>field decl.</td>
</tr>
<tr>
<td>stmt ::= s_{ext}</td>
<td>stmt; stmt</td>
</tr>
<tr>
<td>s_{ext} ::= x : T</td>
<td>x := e</td>
</tr>
<tr>
<td>s_{ext} ::= c_{in}</td>
<td>c_{out}</td>
</tr>
<tr>
<td>c_{in} ::= {new} (this : T)?m(t : thread, x : T, \ldots, x : T) where (e_{bool})</td>
<td></td>
</tr>
<tr>
<td>c_{out} ::= {new} clm(t, v, \ldots, v)</td>
<td></td>
</tr>
<tr>
<td>r_{in} ::= x=t?return(x : T) where (e_{bool})</td>
<td></td>
</tr>
<tr>
<td>r_{out} ::= t!return(e)</td>
<td></td>
</tr>
<tr>
<td>c ::= x</td>
<td>T</td>
</tr>
<tr>
<td>v ::= x</td>
<td>consts</td>
</tr>
</tbody>
</table>

Table 2. Abstract syntax: Multi-threading

structure of incoming and outgoing communications. This gives rise to the abstract syntax of Table 2.

As an illustrating example of the new aspects of the extended language consider the following specification fragment.

new !C(t_{main});
(x := t_{main}?return(y : C) where (y = t);
t!return())

We left out here the initialization and the specification of the provided and required interface, since, apart from the fact that the component provides a subclass \(C\) of the Java class \text{Thread}, it is not important. Thus,
the creation of a new instance of $C$ entails also the creation of a new thread. The main thread $t_{\text{main}}$ starts within the tester and creates a new instance of $C$. After that but before the call returns, the tester expects an incoming call by a new thread $t$. Then the return of the constructor call is expected which yields the new objects. And finally, the thread $t$ returns to the component.

The code generated describes the behavior of the main thread of the tester which controls outgoing method calls and returns and checks the incoming method calls and returns by means of delegation. Due to space limitations, we omit the details of the code generation.

5 Conclusion

In this paper we presented a Java-like test specification language. From specifications of that language we generate automatically Java test drivers which in turn test Java components.

5.1 Future work

Currently, we are implementing a tool for the execution of test specifications of Java programs to demonstrate the feasibility of our framework. As future work, we want to extend the specification language to support other concepts of Java such as monitors and cloning. For these concepts we have already extended our underlying formal framework. Moreover, we plan to provide modularity in our language to allow for the reuse of test-patterns. We also want to provide support to mechanically check the correctness proofs of the code generation. In the multi-threaded case, this requires a formalization of the possible behaviors which arise because of different interleavings of the executing threads. Another promising extension of our testing framework could be to provide an automatic synthesis of specifications of our specification language from higher level specifications like automata or message sequence charts.

5.2 Related work

The framework of this paper is based on a fully abstract semantics for may-testing [6]. Roughly speaking, a denotational semantics is fully abstract w.r.t. testing semantics if it equates exactly those programs that pass the (possibly infinite) set of tests. For Java-like components, fully abstraction results have been exploited in [14,2]. A consequence of full
abstraction is the definability property stating that a sequence of messages is in the semantics of a component if and only if one can construct a successful test scenario for it.

A well-known standard test specification language describing sequences of events is TTCN-3 [1]. It differs in three aspects from our approach. As our test specification language is tailored towards a specific programming language, namely Java, we can faithfully represent the underlying interaction mechanism of the programming language without the need of an additional communication layer (e.g., ports). As a consequence, we have one semantic framework for both the test specification language and the programming language. This uniform formal semantics critically simplifies the correctness of the code generation via a formally established simulation relation. Especially in the multi-threading case, in our experience, the complexity of the code generation, which involves sophisticated synchronizations between the tester and the component, requires a correctness proof and, hence, a uniform formal framework. Thirdly, an interesting problem of test specification is to avoid specifying unrealizable interaction scenarios. Concentrating on a specific concurrency model allows, for instance, in the single threaded case of Java, that only proper nested calls and returns can be tested for.

Message sequence charts (MSC) are a graphical specification language used for the generation of test cases [10]. However the focus of MSC is on timed order of message exchanges and often many test suite details are hidden, like expression evaluation and data generation. This distinguishes from our approach where a test suite is specified in more detail.

In [11] mock objects have been proposed for unit testing. They have been employed in test-driven software development [3]. Based on the paradigm of mock objects, [5] describe [Mock] [7], an open-source library for mocking interfaces in Java. Like the methodology presented here, the external behavior of an object is considered in terms of message sends and returns and in particular distinguishes between the notion of provided and required interfaces. Mock objects are used to mimic the environment of the object under test. In that sense they act as stubs. However, mock objects contain code and assertions to test for the properties of interest, expressed in terms of stimulus-response using message calls and returns. The has been adapted for other object-oriented languages, as well, notably mockpp for C++ [12].

Conceptually related to our and the mock object approach is the testing and validation framework of [8]. Also there, sets of objects are validated in a black-box manner. Sets of objects exchange messages at the
interface of a surrounding environment, which behaves according to a behavioral interface specification, designating the allowed and expected interaction sequences. Unlike the work presented here, the validation framework deals with concurrent, active objects in the Creol language. Furthermore, the behavioral interface specification and the objects under test are both represented in the rewriting system Maude as a common simulation platform, whereas we translate our interface specifications into executable Java code.

References

1. Methods for testing and specification (mts) the testing and test control notation version 3. European Standard ETSI ES 201 8731 v2.2.1, 2002.