Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 243 (2009) 33–47
www.elsevier.com/locate/entcs

Java Test Driver Generation from Object-Oriented Interaction Traces

Frank S. de Boer[1,2]

CWI, Amsterdam, The Netherlands

Marcello B. Bonsangue[1,3] Andreas Grüner[1,4]

LIACS, Leiden, The Netherlands

Martin Steffen[1,5]

UiO, Oslo, Norway

Abstract

In the context of test-driven development for object-oriented programs, mock objects are increasingly used for unit testing. Several *Java* mock object frameworks exist, which all have in common that mock objects, realizing the test environment, are directly specified at the *Java* program level. Though using directly the programming language may facilitate acceptance by software developers at first sight, the entailed syntax noise sometimes distracts from the actual test specification, speaking about interaction traces.
We propose a *Java*-like test specification language, which allows to describe the behavior of the test harness in terms of the expected interaction *traces* between the program and its environment. 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, i.e., a set of classes for mock objects, is *synthesized*.
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: mock objects, black-box testing, Java, trace-based observable behavior, test-driver generation.

[1] Part of this work has supported by the NWO/DFG project Mobi-J (RO 1122/9-4) and by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services.* For more information, see http://credo.cwi.nl

[2] Email: F.S.de.Boer@cwi.nl

[3] Email: marcello@liacs.nl

[4] Email: agruner@liacs.nl

[5] Email: msteffen@ifi.uio.no

doi:10.1016/j.entcs.2009.07.004

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 [19]. To manage the complexity of modern software, 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 [11].

In [17], *mock objects* have been proposed for unit testing. They have been employed in test-driven software development [3]. 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, but contain code and assertions to test for the properties of interest, expressed in terms of stimulus-response using message calls and returns. For *Java*, several mock-object frameworks exist [14] [10] [20]. To lower the burden for software developers, in the above mentioned frameworks mock objects are set up in terms of pure *Java* code. As a consequence, these frameworks have to struggle with syntax noise which may detract from the actual specification.

The contribution of this paper is threefold.

(i) We define a formal specification language for test cases of *Java* components. This specification language is a hybrid of (a subset of) the *Java* language with additional constructs for specification purpose, in particular for formalizing expectations. More specifically, a specification is represented by sequences of incoming (expected) and outgoing calls and returns, using the usual control flow operations like while-loops and conditionals, etc. Further it contains constructs for data manipulation, expression evaluation, and switch on incoming method calls. We think that this novel approach of enriching the programming language with specification constructs allows for both, acceptance by software developers and a concise and more abstract formalization of test cases by means of interaction traces.

(ii) We show how to transform a specification of our test specification language into code of *Java* classes whose instances represent mock objects. A test specification is, thus, executable in a common *Java* environment. We will point out that due to differences in variable scoping and flow of control between a specification and its corresponding *Java* program the code generation is surprisingly complex.

(iii) We identify a certain type of faulty test cases, i.e., test cases which cannot be passed by any possible implementation of the component under test. We show how to distinguish these test cases from faulty components under test at runtime.

We consider a *Java* component to be a set of classes and their instances. Compo-

nents may only communicate via method calls. In particular, fields can only be accessed by objects of the same component. This restricted access is enforced with *Java* access modifiers. To avoid encapsulation problems, inheritance is not allowed to cross the border between *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 can be compositionally described by sequences of method calls and method returns between the component and its environment [1]. 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 component under test (CUT).

The specification language and the code generation are illustrated by an example, realizing a *voting system.* The voting system is a 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 faulty.

The paper is structured as follows. Section 2 defines 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 related and future work.

2 The test specification language for single threaded *Java*

This section formalizes the specification language for the observer's traces. We start with the abstract syntax of the language; Section 2.2 afterwards deals with failure reports.

2.1 Syntax

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

Listing 1: Specification of the voting example

```
1   import java.util.HashSet
2
3   provided Voter {
4                vote() : bool;
5                fVote : bool;
6     }
7
8   required Census {
9                census(HashSet) : bool;
10    }
11
12  c : Census;
13  called : HashSet = new HashSet();
14  voters : HashSet = input.read;
15  conj : bool = true;
16
17  new !Census()! {
18    c:=?return()
19  };
20  c!census(voters.clone()) {
21    while (called.size() < voters.size()) do {
22      (this : Voter)?vote() where (called.contains(this) = false) {
23        called.add(this);
24        conj:=conj & this.fVote;
25        !return(this.fVote)!;
26      } }
27    x:=?return(y : bool) where (y = conj)
28  }
```

In general, a test specification *spec* starts with an (optional) list of imported classes. These classes may only be used internally in the test, but must not contribute to the communication with the CUT. In our example, we import *Java*'s `HashSet` to store the set of voters. The imported classes are omitted from the code in the rest of the paper. It is followed by a list provided of class signatures designating the classes belonging to the environment. In our case, the signature of the environment is specified by the class *Voter* which contains a boolean field *fVote* and its corresponding get-method *vote*. On the other hand, required denotes a non-empty list of class signatures designating the CUT. As explained in the introduction, the fields of the provided-classes cannot be accessed by the CUT and the required-classes only specify method signatures. Finally, *stmt* describes the behavior of the tester and the expected behavior of the component. In our example, it first declares the local variables, resp., initializes them appropriately.

After initialization, the actual interaction is described. First, a new instance of the component class *Census* is created by calling its constructor method and waiting for the return value which is assigned to the local variable *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 *census* of component object *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. To calculate

$spec$	$::= [\mathsf{import}\ ci^*]\ \ \mathsf{provided}\ ci^*\ \ \mathsf{required}\ ci^+\ \ stmt$	spec
ci	$::= c\{meth^*\ \ field^*\}$	class intf.
$meth$	$::= [static]\ \ m(T,\ldots,T) : T$	meth. sign.
$field$	$::= f : T$	field decl.
$stmt$	$::= s_{out} \mid s_{in}$	statements
s_{int}	$::= x : T \mid x := e \mid e.f := e \mid \mathsf{assert}(e_{bool}) \mid x.m(v,\ldots,v)$	internal stmts
s_{in}	$::= skip \mid s_{in}; s_{in} \mid c_{in} \mid \mathsf{callswitch}:\ c_{in}^+$	stmts in
	$\mid \mathsf{if}\ (e_{bool})\ \mathsf{then}\ \ s_{in}\ \ \mathsf{else}\ \ s_{in} \mid \mathsf{while}\ (e_{bool})\ \mathsf{do}\ \ s_{in}$	
c_{in}	$::= (this{:}T)?m(x{:}T,\ldots,x{:}T)\ \ \mathsf{where}\ \ e_{bool}\ \ s_{out}\ \ !\mathsf{return}(e)$	incoming call
s_{out}	$::= skip \mid s_{out}; s_{out} \mid c_{out} \mid \mathsf{new}\ \ c_{out} \mid s_{int}$	stmts out
	$\mid \mathsf{if}\ (e_{bool})\ \mathsf{then}\ \ s_{out}\ \ \mathsf{else}\, s_{out} \mid \mathsf{while}\ (e_{bool})\ \mathsf{do}\ \ s_{out}$	
c_{out}	$::= c!m(v,\ldots,v)\ \ s_{in}\ \ x{=}?\mathsf{return}(x : T)\ \ \mathsf{where}\ \ e_{bool}$	outgoing call
c	$::= x \mid T$	callee
v	$::= x \mid consts$	values

Table 1
Abstract syntax

the conjunction, the component has to find out the votes of the voter objects of set *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, *census* returns to the tester the final result.

More specifically, *after* the call of the component's method *census* and *before* the corresponding return with the final result, the component must call method *vote* of each *Voter* object which belong to the tester. Only then the invocation of *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 begin-

ning and end of a block construct.[6] 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 *Java* code for such specifications.

2.2 Failure reports

A terminating execution of a test specification is successful if not generating a failure report. In general, failures are caused by violated 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.

new $!C()$ (this : $T)?m()$ ... $x :=$?return()

This example can give rise to two kinds of failures. First, the constructor of component class C returns without calling any method of the tester. Second, the constructor calls a different, unexpected method of the tester. The implementation of the specification language generates automatically appropriate failure reports.

Note that it is possible to specify tests which always fail because of unrealizable interaction scenarios. E.g., consider

$x :=$ new $T()$; new $!C()$ { (this:$T)?m(u{:}T)$ where $(u = x)$ ... }

which, after instantiating a test class T, calls the constructor of component class C and then expects an incoming call with the newly created instance of T as parameter. As the instantiation of a tester class by the tester itself is an internal action this newly created object is unknown to the component unless the tester communicates it. Of particular interest are thus techniques which identify such failures. As it is impossible to identify all such failures statically, we provide a runtime check by recording all identities of tester objects known to the component. If a test execution leads to a situation where a tester object, which actually cannot be known by the component, is expected in an incoming communication, the tester reports an invalid test and aborts.

[6] We consider a call of a constructor as a special method call.

3 Code generation

This section describes how to generate *Java* code from a test specification for the methods of the tester classes. To understand the general strategy for the generation, it is useful to clarify the nature of the specification language and especially, what are the differences to (or additions to) a programming language like *Java*. The abstract goal of the specification language is the specification of interaction *traces* used for testing and employing programming-like structuring such as classes and mentioning methods. As far as the *interaction* is concerned, i.e., the calls and returns exchanged at the interface of the CUT, there is a strong duality between *incoming* and *outgoing* communication, seen from the perspective of the tester (cf. Table 1). Outgoing calls and returns must be *realized* by the tester, and incoming communication must be *checked* by it, and both adhering to the *linear order* as given by the specification language, specifying a set of traces. It suggests itself, to realize the interaction labels as given on the *specification* level by corresponding method calls and returns at the *program* level. To do so requires to tackle the following two points:

control flow: The code at the *Java*-level must be contained in bodies of methods, corresponding to the *incoming* method-labels of the trace specification, i.e., the test-code must be appropriately "distributed" over different method bodies and classes. Furthermore and as mentioned, the order of accepting incoming communications and generating outgoing ones must be realized as given by the specification. We use a labeling-mechanism to assure proper interaction *sequencing.*

variable binding: The parameters mentioned in an incoming communication at the specification level introduce a *scope* much as a method declaration at the program level introduces a scope, namely for the method's formal parameters. There is a crucial difference between the two scoping mechanisms. At the program label, the formal parameters' scope clearly spans the method body, only. In contrast, at the specification level, the scope can extend beyond the body of the corresponding method, as given by the trace specification.

These are, apart from technicalities, the two main differences to be bridged by the translation. 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 results in the specification of Listing 2 (in the code, we omit the "preamble" of the import statements and the declaration of the required and provided interface, as that part coincides with the code of Listing 1).

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 $next := \ell$ where *next* is a new auxiliary variable for the tester and ℓ is a label. Referring to the next expected *incoming* call or return, *next* is updated right before each *outgoing* call or return. Thus, the result of this preprocessing step is that all internal computations fall within the scope of the control of the tester.

Listing 2: Preprocessed specification

```
1
2   c : Census;
3   called : HashSet = new HashSet();
4   voters : HashSet = input.read;
5   conj : bool = true;
6
7   new !Census() {
8     [ℓ0]c:=?return()
9   };
10  if (called.size() < voters.size()) then next:=ℓ1 else next:=ℓ2;
11  c!census(voters.clone())! {
12    [ℓ1](this : Voter)?vote() where (called.contains(this) = false) {
13      called.add(this);
14      conj:=conj & this.fVote;
15      if (called.size() < voters.size()) then next:=ℓ1 else next:=ℓ2;
16      !return(this.fVote)!;
17    }
18    [ℓ2] x:=?return(y : bool) where (y = conj)
19  }
```

The preprocessing step can be formally described by two mutually recursively applied functions $prep_{in} : s_{in} \times stmt \rightarrow s_{in} \times stmt$; and $prep_{out} : s_{out} \rightarrow s_{out}$. Both functions basically expect an s_{in}, resp., an s_{out}, statement and return a corresponding statement which is free from s_{in}-while-loops but annotated with labels. Additionally, as a second argument $prep_{in}$ expects a *next* update statement describing the next expected incoming call or return that *follows* the s_{in} statement. This update statement is inserted in s_{in} in front of its last outgoing return. Dually, $prep_{in}$ yields a *next* update statement which describes the next expected incoming call of s_{in} itself. The definition of $prep_{out}$ is straightforward. Its solely interesting case deals with an outgoing call:

$$prep_{out}(!c\ s_{in}\ ?r) = s_{nxt};\ !c\ s'_{in}\ [l]?r \quad \text{where } (s'_{in}, s_{nxt}) = prep_{in}(s_{in}, next := l)\ .$$

Note, that $!c$ and $?r$ are used here as abbreviations for an outgoing call and its corresponding incoming return. Moreover, l is a new label and the assignment $next := l$ says that the next expected incoming communication is the incoming return labeled with l. The statement s_{nxt} describes the expectation of the next incoming call according to s_{in}.

The definition of $prep_{in}$ is as follows.

$$prep_{in}(?c\ s_{out}\ !r, s_{nxt}) = ([l]?c\ prep_{out}(s_{out});\ s_{nxt}\ !r, \mathsf{next} := l)$$

$$prep_{in}(s^1_{in}; s^2_{in}, s_{nxt}) = (s'^1_{in}; s'^2_{in}, s^1_{nxt}) \text{ where}$$
$$(s'^2_{in}, s^2_{nxt}) = prep_{in}(s^2_{in}, s_{nxt}) \text{ and } (s'^1_{in}, s^1_{nxt}) = prep_{in}(s^1_{in}, s^2_{nxt})$$

$$prep_{in}(\mathsf{while}(b)\,\mathsf{do}\,s_{in}) = (s^2_{in}, \mathsf{if}(b)\,\mathsf{then}\,s^1_{nxt}\,\mathsf{else}\,s_{nxt}) \text{ where}$$
$$(s^1_{in}, s^1_{nxt}) = prep_{in}(s_{in}, s_{nxt}) \text{ and}$$
$$(s^2_{in}, s^2_{nxt}) = prep_{in}(s_{in}, \mathsf{if}(b)\,\mathsf{then}\,s^1_{nxt}\,\mathsf{else}\,s_{nxt})$$

$$prep_{in}(skip, s_{nxt}) = (skip, s_{nxt})$$

$$prep_{in}(\mathsf{if}(b)\,\mathsf{then}\,s^1_{in}\,\mathsf{else}\,s^2_{in}, s_{nxt}) = (s'^1_{in}; s'^2_{in}, \mathsf{if}(b)\,\mathsf{then}\,s^1_{nxt}\,\mathsf{else}\,s^2_{nxt}) \text{ where}$$
$$(s'^1_{in}, s^1_{nxt}) = prep_{in}(s^1_{in}, s_{nxt}) \text{ and } (s'^2_{in}, s^2_{nxt}) = prep_{in}(s^2_{in}, s_{nxt})$$

A sequential composition of two s_{in} statements is transformed by first transforming the second statement which processes the expectation s_{nxt} of the incoming communication that follows the composition. Moreover, it yields its own expectation s^2_{nxt} which in turn is processed in the first s_{in} statement. In order to transform an s_{in}-while-loop, $prep_{in}$ is called for the body of the while loop. Its resulting expectation s^1_{nxt} is used to formulate the expectation for the entrance of the while-loop and also for the loop itself. For the latter, the body of the while-loop is transformed a second time.

Listing 3 contains the code 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*. According to the test specification, the tester starts the execution. Thus, a main method is introduced to describe its behavior, 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*. The implementation of the method *voter*, on the other hand, checks whether it has been called when expected. This expectation is expressed by a test involving the corresponding label.

In the example, the method *vote* is called only once for every *Voter* object. In general, however, a method can of course be called several times in different situations and with different reactions. We explain the general case in terms of the following two fragments of a preprocessed test specification. The overall structure of the fragment in Listing 4 depicts n occurrences of incoming calls of the method m. Furthermore, the ith incoming call contains an outgoing call to m' at top level. Finally, this latter outgoing call contains an incoming call to a method m''. This leads to the *Java* code fragment of method m, shown in Listing 5.

The method body consists of a `switch` dispatching on the occurrence of the incoming call. The code for each such case is generated from the corresponding

Listing 3: Voting example: *Java* code

```
1   class Tester {
2     enum Label = {ℓ0, ℓ1, ℓ2};
3     Label next = ℓ0;
4     HashSet called = new HashSet();
5     HashSet voters = new input.read;
6     bool conj = true;
7     Census c;
8
9     void main() {
10      c = new Census();
11      assert(next == ℓ0);
12      if (called.size() < voters.size()) Tester.next = ℓ1;
13      else Tester.next = ℓ2;
14      y = c.census(voters);
15      assert(next == ell2);
16      assert(y == conj);
17      x = y;
18    } }
19
20  class Voter {
21    bool vote() {
22      switch(Tester.next) {
23        case ℓ1: {
24          assert(called.contains(this) == false);
25          called.add(this};
26          conj=conj & this.fVote;
27          if (called}.size() < voters.size()) Tester.next = ℓ1;
28          else Tester.next = ℓ2;
29          return(this.fVote);
30        }
31        case default:
32          assert(false);
33      } } }
```

Listing 4: Specification fragment: the general case

```
⋮
[ℓ1] (this : T)?m(...) ...
⋮
[ℓi] (this : T)?m(...) {
  ⋮
  x!m′() {
    ⋮
    [ℓ] (this : T′)?m″() ...
    ⋮
    [ℓ′] ?return ...
  } }
⋮
[ℓn] (this : T)?m(...) ...
```

occurrence. E.g., the outgoing call $x!m'$ is preceded with the update of *next* with label ℓ 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.

Listing 5: *Java* code fragment: the general case

```
T m(...) {
  switch(Tester.next) {
    case ℓ1: ...
    ⋮
    case ℓi: {
      next = ℓ;
      x.m′();
      assert(next == ℓ′);
      next = ...
      return e
    }
    ⋮
    case ℓn: ...
```

Listing 6: specification fragment: formal parameters

```
[ℓ] (this : T)?m(x : T′) {
  ⋮
  [ℓ′] (this : T)?m(y : T′) {
    if (x < y)
    then this.f!m′(x) ...
    else this.f!m′(y) ...
  }
}
```

Listing 7: *Java* code fragment: formal parameters

```
T″ m(T′ n) {
  switch(Tester.next) {
    case ℓ1: Tester.ℓ_x = n; ...
    case ℓ2: {
      Tester.ℓ′_y = n;
      if(Tester.ℓ_x < Tester.ℓ′_y)
            this.f.m′(Tester.ℓ_x); ...
      else this.f.m′(Tester.ℓ′_y); ...
    } } }
```

The second complication in the code generation is that the formal parameters of incoming method calls in the specification language are different from the formal parameters of *Java* method definitions and thus cannot be directly translated. To understand this, consider the following fragment of another preprocessed specification presented in Listing 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. To describe the scope of the variables we annotate them with the label of their occurrence. For the above specification, the implementation fragment is given in Listing 7.

$spec$	::= $[\mathsf{import}\ clintf^*]\ \mathsf{provided}\ clintf^+\ \mathsf{required}\ clintf^+\ stmt$	spec
$clintf$	::= $c : c\ \{meth^*\ field^*\}$	class intf.
$meth$	::= $[static]\ m(T,\ldots,T) : T$	meth. sign.
$field$	::= $[static]\ f : T$	field decl.
$stmt$	::= $s_{ext} \mid s_{int} \mid stmt; stmt \mid \mathsf{if}\ (e_{bool})\ \mathsf{then}\ stmt\ \mathsf{else}\ stmt$	
	$\mid \mathsf{while}\ (e_{bool})\ \mathsf{do}\ stmt \mid \mathsf{callswitch}\colon (c_{in} :\ stmt)^+$	statements
s_{int}	::= $x : T \mid x := e \mid e.f := e \mid \mathsf{assert}(e_{bool}) \mid x.m(v,\ldots,v)$	internal stmts
s_{ext}	::= $c_{in} \mid c_{out} \mid r_{in} \mid r_{out}$	stmts ext
c_{in}	::= $[\mathsf{new}]\ (this : T)?m(t : thread, x : T,\ldots,x : T)\ \mathsf{where}\ e_{bool}$	incoming call
c_{out}	::= $[\mathsf{new}]\ c!m(t,v,\ldots,v)$	
r_{in}	::= $x{=}t?\,\mathsf{return}(x : T)\ \mathsf{where}\ e_{bool}$	
r_{out}	::= $t!\,\mathsf{return}(e)$	
c	::= $x \mid T$	callee
v	::= $x \mid consts$	values

Table 2
Abstract syntax: Multi-threading

4 Generalizing to the multi-threaded case

To extend the specification language to a multi-threaded setting, we first extend the communications between the component and the tester with an additional parameter representing the executing thread. Second, we need to relax the nested block structure of incoming and outgoing communications. This gives rise to the abstract syntax of Table 2.

For illustration of the new aspects, consider the following specification fragment.

```
...
new !C(t_main);
(this : T)?m(t : Thread);
x := t_main?return(y : C) where (y = t);
t !return()
```

We left out again initialization and the provided and required interface. The creation of a new instance of C entails also the creation of a new thread. The main thread t_{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 yielding the new objects. Finally, the thread returns to the component. The generated code describes the behavior of the tester's main thread which conrols outgoing method calls and returns and checks

the incoming communication by means of delegation. For lack of space, we omit the details of the code generation.

5 Conclusion

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 plan 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 specification like automata or message sequence charts.

5.2 Related work

The presented framework is based on a fully abstract semantics for may-testing [13]. Roughly speaking, a denotational semantics is fully abstract wrt. a testing semantics if it equates exactly those programs that pass the same (possibly infinite) set of tests. For *Java*-like components, full abstraction results have been explored in [2,1]. A consequence of full abstraction is the definability property stating that a sequence of messages is in the semantics of a component iff one can construct a successful test scenario for it.

Various approaches for testing especially object-oriented systems have been developed. Testing for especially concurrent object-oriented programs based on synchronization sequences is investigated in [9], based on Petri nets and OBJ (OBJSA [6] [5]). An overview over various integration testing approaches for object-oriented systems in [8].

A well-known standard test specification language based on sequences of events is TTCN-3 [21]. 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 properly nested calls and returns can be tested for. Message sequence charts (MSC) are a graphical specification formalism used for the generation of test cases [16]. The focus of MSCs, however, is on the timed order of message exchanges and often many test suite details are hidden, like expression evaluation and data generation. This differs from our approach where a test suite is specified in more detail. [4] proposes a specification-based (i.e., black-box) testing method for object-oriented software. The desired interface behavior is described in the object-oriented specification language *CO-OPN/2*, which is formally based on Petri nets and a transition-system semantics. The approach can be seen as a generalization to the one of [7], [18] (used for testing abstract data types) to deal with object-oriented programs. Sequences of the interface behavior can be described using a simple modal logic known as *Hennessy-Milner logic* (HML) and hence, the corresponding notion of observational equivalence is bisimulation equivalence.

Conceptually related to our and the mock object approach is the testing and validation framework of [15]. 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. Also in the context of *Creol*, [12] investigate a simple trace specfication language with a focus on the asynchronous nature of that communication model and exploiting the dual nature of interaction with a (concurrent) object: outgoing communication is being tested for complies, whereas incoming communication is scheduled.

References

[1] E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen. Object connectivity and full abstraction for a concurrent calculus of classes. In Z. Li and K. Araki, editors, *ICTAC'04*, volume 3407 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag, July 2004.

[2] E. Ábrahám, F. S. de Boer, M. M. Bonsangue, A. Grüner, and M. Steffen. Observability, connectivity, and replay in a sequential calculus of classes. In M. Bonsangue, F. S. de Boer, W.-P. de Roever, and S. Graf, editors, *Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 296–316. Springer-Verlag, 2005.

[3] D. Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall, 2003.

[4] S. Barbey, D. Buchs, and C. Péraire. A theory of specification-based testing for object-oriented software. In *Proccedings of the European Dependable Computing Conference*, volume 1150 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[5] E. Battiston, A. Chizzoni, and F. D. Cindio. Clown as a testbed for concurrent object-oriented concepts. In *Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri-Nets*, Lecture Notes in Computer Science, pages 131–163. Springer-Verlag, 2001.

[6] E. Battiston, F. de Cindio, and G. Mauri. Modular algebraic nets to specify concurrent systems. *IEEE Transactions in Software Engineering*, 22(10):689–705, 1996.

[7] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications. *IEEE Software Engineering Journal*, 6(6):387–405, Nov. 1991.

[8] W. K. Chan, T. Y. Chen, and T. H. Tse. An overview of integration testing techniques for object-oriented programs. In *Proceedings of the 2nd ACIS Annual International Conference on Computer and Information Science (ICIS 2002)*, 2002.

[9] H. Y. Chen, Y. X. Sun, and T. H. Tse. A strategy for selecting synchronization sequences to test concurrent object-oriented software. In *Proceedings of the 27th International Computer Software and Application Conference (COMPSAC 2003), Los Angeles, California*. IEEE Computer Science Press, 2003.

[10] EasyMock. http://www.easymock.org, 2007.

[11] M. Fewster and D. Graham. *Software Test Automation*. Addison Wesley, 1999.

[12] I. Grabe, M. Steffen, and A. B. Torjusen. Executable interface specifications for testing asynchronous creol components. Technical Report 375, University of Oslo, Dept. of Computer Science, July 2008. A shorter version has been submitted for conference proceedings.

[13] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[14] jMock. www.jmock.org, 2007.

[15] E. B. Johnsen, O. Owe, and A. B. Torjusen. Validating behavioral component interfaces in rewriting logic. *Fundamenta Informaticae*, 82(4):341–359, 2008.

[16] B. Koch. *Test Purpose Based Test Generation for Distributed Test Architectures*. Phd thesis, Universität zu Lübeck, Germany, 2001.

[17] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: Unit testing with mock objects. In G. Succi and M. Marchesi, editors, *Extreme Programming Examined*, The XP Series, pages 287–301. Addison-Wesley, 2001.

[18] B. Marre. *Sélection automatique de jeux de tests à partir de spécification algébraiques en utilisant la programmation logique*. PhD thesis, Université de Paris XI, Jan. 1991.

[19] R. Patton. *Software Testing*. SAMS, second edition, July 2005.

[20] rMock. http://rmock.sourceforge.net, 2007.

[21] Methods for testing and specification (mts). The testing and test control notation version 3 (ttcn-3). European Standard ETSI ES 201 8731 v2.2.1, 2002.