Viewpoint-based testing of concurrent components — Source link ⧉

Luke Wildman, Roger Duke, Paul Strooper

Institutions: University of Queensland

Topics: Test strategy, Test Template Framework, Viewpoints, Formal specification and Concurrency

Related papers:

- Strategies for consistency checking based on unification

- System design validation using formal models

- Formally Justifying User-Centred Design Rules: A Case Study on Post-completion Errors

- A methodology for developing component-based agent systems focusing on component quality

- Supporting scenario-based requirements engineering

Viewpoint-Based Testing of Concurrent Components

Luke Wildman, Roger Duke, and Paul Strooper

School of Information Technology and Electrical Engineering,
The University of Queensland,
{luke,rduke,pstroop}@itee.uq.edu.au,
Fax: +61 7 3365 4999,
Phone: +61 7 3365 2097

Abstract. The use of multiple partial viewpoints is recommended for specification. We believe they also can be useful for devising strategies for testing. In this paper, we use Object-Z to formally specify concurrent Java components from viewpoints based on the separation of application and synchronisation concerns inherent in Java monitors. We then use the Test-Template Framework on the Object-Z viewpoints to devise a strategy for testing the components. When combining the test templates for the different viewpoints we focus on the observable behaviour of the application to systematically derive a practical testing strategy. The Producer-Consumer and Readers-Writers problems are considered as case studies.

Keywords: Viewpoints, Object-Z, Test Template Framework, Concurrency, Java

1 Introduction

Concurrent programs are notoriously difficult to test because of the ways in which threads can synchronise and interact with each other. In this paper, we focus on testing concurrent Java components and we assume that the component can be accessed by any number of threads. We apply a specification-based testing approach to derive a strategy for testing such concurrent components. When devising a strategy for testing concurrent components, one has to assume a basic concurrency model whether it is the Java monitors model (as in our case) or another (more generic) model. Our starting point for deriving a testing strategy is a formal, Object-Z [7] model of a Java monitor. We apply the Test Template Framework [19] to this model to derive generic test conditions for Java monitors. However, the strategy will be applied to specific components, so we need to consider what happens when we combine this information with application-specific test conditions. To do this, we take a viewpoints-based approach. We combine the generic test conditions from the Java monitor model with test conditions derived from an Object-Z specification of a model of the application we want to test. As we illustrate using a Producer-Consumer monitor,

if we do this naively, then the number of test conditions becomes unmanageable. That is, if we model the Producer-Consumer application in Object-Z, apply the Test Template Framework, and combine the resulting test conditions with the generic ones, we obtain a very large number of test conditions. Moreover, many of these conditions test the generic behaviour of Java monitors, rather than the application-specific behaviour of the Producer-Consumer monitor that we are interested in.

To alleviate this problem, we use a restricted viewpoint of the general model of Java monitors that focuses on those aspects of the model that have an externally visible effect on the behaviour of the concurrent component. By combining the test conditions from this restricted viewpoint with the application-specific test conditions, we reduce the number of test conditions and can focus on those conditions that are relevant to the observable behaviour of the application.

Although we do not discuss it in detail, the test conditions that are generated by the approach described in this paper could be used to generate test sequences in a testing tool such as ConAn [12].

1.1 Related Work

We apply the Test Template Framework [19] to concurrent Java components in Object-Z [7]. Test Templates have been generated from Object-Z before [5] and there has been some work dealing with the issues of concurrency under the guise of interactive systems [15]. Interactive systems have also been considered by others [1]. However, the Test Template Framework has not previously been applied directly to concurrent components.

The use of the Test Template Framework with inheritance has been considered elsewhere [17], as has the combination of Object-Z operations [18]. However, neither deal with the issues of multiple inheritance as we do here.

We use viewpoint-based specifications [10,3]. That is, multiple partial specification are used rather than a single monolithic specification. This approach allows different concerns to be treated separately without constructing a complicated unified model. Viewpoints-based testing has been considered before [14, 4], but in this previous work different viewpoints are represented in different specification languages, whereas we use a single specification language.

1.2 Overview

An introduction to Java and the Java concurrency model is given in Section 2. This includes a formal specification of a Java monitor in Object-Z. In Section 3 the Test Template Framework is described and applied to the Java monitor specification to produce a set of test conditions. The specification of the Producer-Consumer monitor is considered in Section 4 along with test conditions for the application-specific behaviour. In Section 5 a naive combination of the test hierarchies from the two viewpoints is discussed and a different approach based on a restricted concurrency viewpoint is presented. To further evaluate and illustrate

the approach, its application to the Readers-Writers problem is considered in Section 6. Concluding remarks are presented in Section 7.

2   Java Concurrency

A typical application involving Java concurrency is presented in Figure 1. The Java code implements a finite buffer which may be shared between many Producer and Consumer threads for the purpose of communicating resources. We shall assume a basic understanding of the Java synchronisation model. The `put`

```
public class BufferImpl {

    protected int[] buf;
    protected int in = 0;
    protected int out= 0;
    protected int count= 0;
    protected int size;

    public BufferImpl(int size) {
        this.size = size;
        buf = new int[size];
    }

    public synchronized void put(int i) throws InterruptedException {
        while (count==size) wait();
        buf[in] = i;
        ++count;
        in=(in+1) % size;
        notifyAll();
    }

    public synchronized int get() throws InterruptedException {
        while (count==0) wait();
        int i =buf[out];
        --count;
        out=(out+1) % size;
        notifyAll();
        return (i);
    }
}
```

Fig. 1. A Java implementation of a finite buffer.

method is used to add a resource to the finite buffer. The calling thread waits until space is available and then notifies all waiting threads after it adds the resource. The `get` method retrieves a resource from the buffer. The calling thread

waits until a resource is available and then notifies all other waiting threads after it removes the resource. An object of the class BufferImpl conforms to the underlying Java thread synchronisation model based on synchronised methods and blocks of code, together with the Java methods wait, notify and notifyAll inherited from the Java Object superclass. (Other Java features like thread creation, join, sleep and interrupt, or the deprecated suspend, resume and stop will not be discussed.)

In previous work [8], the specification of the class *Object* (presented below) was used to capture the underlying Java concurrency mechanism consistent with that described within the Java Language Specification [9] and the Java Virtual Machine (JVM) Specification [11]. Instances (i.e. objects) of the class *Object* denote the objects in a Java system, such as instances of BufferImpl; the class *Object* captures the underlying concurrency of the system from the viewpoint of these objects.

In the specification of *Object*, *Thread* denotes the set of all possible program threads in a Java system. The set *OneThread* is defined by

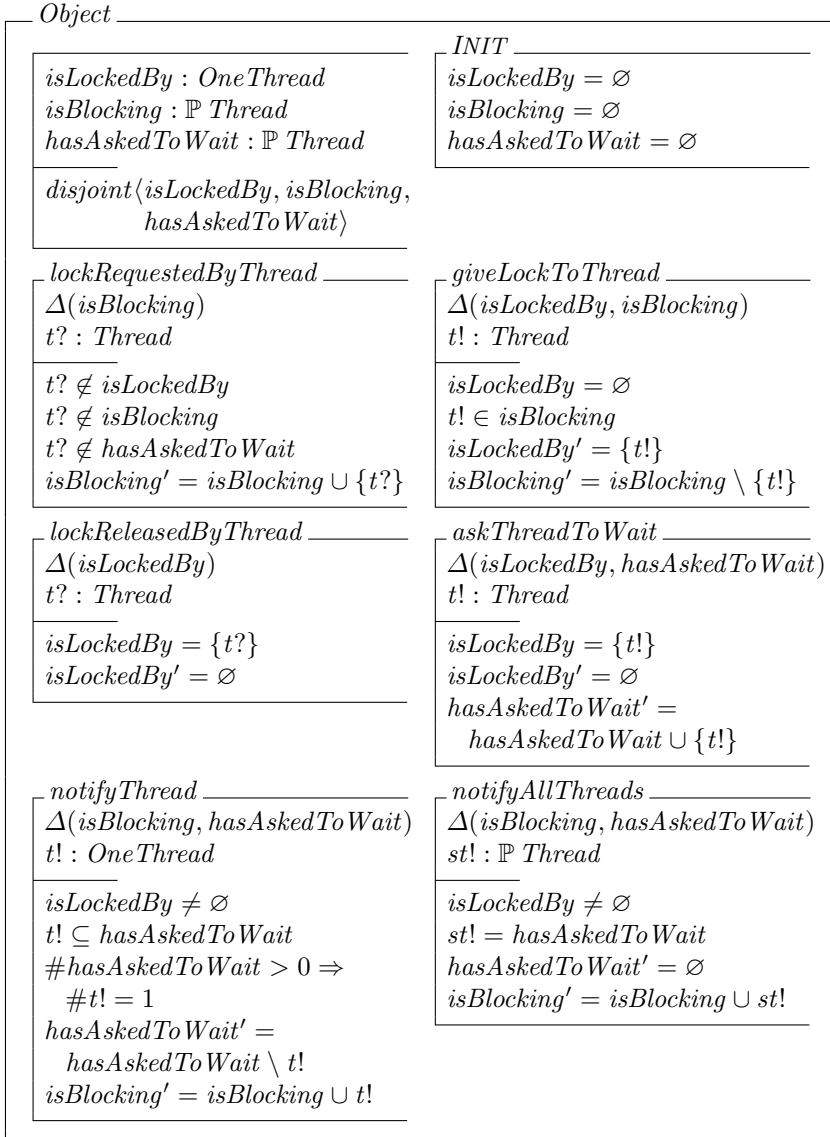$$OneThread == \{st : \mathbb{P} \ Thread \mid \#st \leqslant 1\}$$

and denotes subsets of threads containing at most one thread.

Consider now the class *Object* in detail. A basic understanding of Object-Z is assumed. The three state attributes each denote subsets of threads. The attribute *isLockedBy* denotes the thread (if any) that holds the lock on the object, *isBlocking* denotes the set of threads blocked on the object, and *hasAskedToWait* denotes the set of threads that are waiting to be notified by the object. These three subsets are mutually disjoint, capturing the requirement that at any given time a thread can play at most one of these roles for any given object. Initally all three subsets are empty.

A call to a synchronised method or entry into a synchronised block is modelled in two steps. The thread initiates a *lockRequestedByThread* and the JVM responds with a *giveLockToThread*. The operation *lockRequestedByThread* captures the object's view of the situation when a Java thread seeks to enter a synchronised block of an object it is not already locking. We do not model recursive locks. (If a thread holds the lock on an object, in our model we assume that it can enter any synchronised block of that object without restriction. In particular, it does not need to request a further lock on the object.) The thread in question cannot be blocked on the object and cannot be waiting to be notified by the object. The outcome of the operation is that the thread joins the set of threads blocked by the object.

The operation *giveLockToThread* specifies what happens when the JVM selects a thread from among those currently blocked on the object and allows that thread to lock it. This captures the object's view of the situation when a Java thread is given access to the synchronised blocks of the object. The object cannot already be locked, and the outcome of the operation is that the thread is given the lock on the object and is removed from the set of blocked threads. *giveLockToThread* can occur whenever the object is not locked, i.e., initially, or whenever a thread releases the lock as described by the following 2 operations.

The operation *lockReleasedByThread* specifies what happens when the thread that currently holds the lock on the object releases that lock. This captures the object's view of the situation when the Java thread currently locking the object no longer requires access to any synchronised block of the object. The outcome is that the object is no longer locked.

Object

isLockedBy : *OneThread*
isBlocking : $\mathbb{P}$ *Thread*
hasAskedToWait : $\mathbb{P}$ *Thread*

disjoint⟨*isLockedBy*, *isBlocking*,
         hasAskedToWait⟩

INIT

isLockedBy $= \varnothing$
isBlocking $= \varnothing$
hasAskedToWait $= \varnothing$

lockRequestedByThread

$\Delta(isBlocking)$
t? : *Thread*

t? $\notin$ *isLockedBy*
t? $\notin$ *isBlocking*
t? $\notin$ *hasAskedToWait*
isBlocking' $=$ *isBlocking* $\cup \{t?\}$

giveLockToThread

$\Delta(isLockedBy, isBlocking)$
t! : *Thread*

isLockedBy $= \varnothing$
t! $\in$ *isBlocking*
isLockedBy' $= \{t!\}$
isBlocking' $=$ *isBlocking* $\setminus \{t!\}$

lockReleasedByThread

$\Delta(isLockedBy)$
t? : *Thread*

isLockedBy $= \{t?\}$
isLockedBy' $= \varnothing$

askThreadToWait

$\Delta(isLockedBy, hasAskedToWait)$
t! : *Thread*

isLockedBy $= \{t!\}$
isLockedBy' $= \varnothing$
hasAskedToWait' $=$
  hasAskedToWait $\cup \{t!\}$

notifyThread

$\Delta(isBlocking, hasAskedToWait)$
t! : *OneThread*

isLockedBy $\neq \varnothing$
t! $\subseteq$ *hasAskedToWait*
$\#hasAskedToWait > 0 \Rightarrow$
  $\#t! = 1$
hasAskedToWait' $=$
  hasAskedToWait $\setminus$ *t!*
isBlocking' $=$ *isBlocking* $\cup$ *t!*

notifyAllThreads

$\Delta(isBlocking, hasAskedToWait)$
st! : $\mathbb{P}$ *Thread*

isLockedBy $\neq \varnothing$
st! $=$ *hasAskedToWait*
hasAskedToWait' $= \varnothing$
isBlocking' $=$ *isBlocking* $\cup$ *st!*

The operation *askThreadToWait* specifies what happens when the object requests that the thread currently holding the lock on the object wait for notification. This captures the object's view of the situation when a Java thread

executes a wait while accessing a synchronised block of the object. The outcome is that the thread is added to the set of threads waiting to be notified, and the object is no longer locked.

The operation *notifyThread* specifies what happens when the JVM selects a thread from among those currently waiting on the object and notifies it. This captures the object's view of the situation when a notify is executed by some other thread currently accessing a synchronised block of the object. The outcome is that the selected thread is removed from the set of waiting threads and added to the set of threads blocked on the object. Note that *notifyThread* does nothing if no threads are waiting, i.e. in this case execution of notify will wake no threads.

The operation *notifyAllThreads* is like *notifyThread* except that all threads waiting on the object are notified. This captures the object's view of the situation when a notifyAll is executed by some other thread currently accessing a synchronised block of the object. The outcome is that all the threads currently waiting on the object are added to the set of threads blocked on the object, while the set of threads waiting on the object becomes empty.

3   Object Class Test Cases

3.1   Test Template Framework

The Test Template Framework (TTF) [19] provides a systematic way to select and record abstract test cases for individual operations from a model-based formal specification. Test case selection remains under the control of the human tester and multiple testing strategies are supported. The original work focused on specifications written in the Z notation, but the framework has been extended for Object-Z specifications [5].

3.2   Process

The framework provides a systematic way of finding partitions for each operation. The precondition of an operation is used as the starting point for partitioning since Object-Z operations are disabled for inputs outside the precondition. The precondition is called the *valid input space* (VIS). The valid input space is a subset of the *input space* (IS) of an operation, which is defined as the restriction of the operation's signature to input components (inputs and pre-state components). The *output space* (OS) is similarly defined over the output components.

A *strategy* in the framework identifies a particular technique for deriving test cases. Both traditional techniques such as input partitioning and boundary analysis, and specialised techniques that exploit the specification notation are used. The framework encourages the use of multiple strategies to build the *test template hierarchy* (TTH). The hierarchy captures the relationships between test templates and strategies, and serves as a record of the test development process. The root of the hierarchy is the valid input space.

The hierarchy is created by applying testing strategies to existing test templates to derive additional ones. A test template hierarchy is usually a directed

acyclic graph with the leaf nodes partitioning the valid input space. Strategies are applied to the hierarchy until the tester is satisfied that the leaf templates of the hierarchy represent adequate sources of tests, that is, every instantiation of a leaf template is equally likely to reveal an error in an implementation.

To identify test data, the tester instantiates the leaf test templates (TT) in the test template hierarchy by supplying specific values for the state and inputs. The resulting test templates are called instance templates (IT). The framework also defines output templates (OT) corresponding to test or instance templates. These define the expected final states and outputs corresponding to the test or instance template. The output is calculated by restricting the operation to the input described in the template and then projecting onto the operation's output space.

3.3   Object TTH

We now discuss the application of the Test Template Framework to the *Object* specification. We discuss the derivation for the *lockRequestedByThread* operation in detail for each step and mention the interesting aspects of the application to the other operations.

Note that the *INIT* schema is ignored because it is not an operation. That is, it has no input space to which we can apply the Test Template Framework.

Valid Input Space. The Valid Input Space for *lockRequestedByThread* is:

$$VIS_{lockRequestedByThread} \mathrel{\widehat{=}} \text{pre } lockRequestedByThread$$

which expands and simplifies to

$$VIS_{lockRequestedByThread} \mathrel{\widehat{=}}$$
$$\big[\, isLockedBy : OneThread;\ isBlocking, hasAskedToWait : \mathbb{P}\ Thread;$$
$$t? : Thread \mid \text{disjoint}\langle isLockedBy, isBlocking, hasAskedToWait\rangle \wedge$$
$$t? \notin isLockedBy \wedge t? \notin isBlocking \wedge t? \notin hasAskedToWait \,\big]$$

Test Templates. The *lockRequestedByThread* test template hierarchy is defined as follows (taking the set of all possible strategies as given):

$$[Strategies];$$
$$TTH_{lockRequestedByThread} :$$
$$TT_{lockRequestedByThread} \times Strategy \nrightarrow \mathbb{P}\ TT_{lockRequestedByThread}$$

The function $TTH_{lockRequestedByThread}$ captures the combination of test templates with testing strategies to derive additional test templates. For most strategies, the children partition the parent.

A common, intuitive testing strategy is 0-1-many, based on the cardinality of a set such as *isBlocking*. This strategy is referred to as (one example of) type-based selection. Type-based selection (*TB : Strategies*) is identified as a

particular testing strategy and is used to partition the valid input space into cases where *isBlocking* is empty, a singleton, and a set with more than one element, generating three test templates (distinguished with numeric subscripts) as a result.

$$TT_{lockRequestedByThread.0} \mathrel{\widehat{=}} \left[\, VIS_{lockRequestedByThread} \mid \#isBlocking = 0 \,\right]$$
$$TT_{lockRequestedByThread.1} \mathrel{\widehat{=}} \left[\, VIS_{lockRequestedByThread} \mid \#isBlocking = 1 \,\right]$$
$$TT_{lockRequestedByThread.2} \mathrel{\widehat{=}} \left[\, VIS_{lockRequestedByThread} \mid \#isBlocking > 1 \,\right]$$

$$TTH_{lockRequestedByThread}(VIS_{lockRequestedByThread}, TB) =$$
$$\{ TT_{lockRequestedByThread.0}, TT_{lockRequestedByThread.1}, TT_{lockRequestedByThread.2} \}$$

We can apply a similar type-based selection strategy to both the *isLockedBy* and *hasAskedToWait* state variables to partition the above test templates further. Since *isLockedBy* is of type *OneThread*, it can only be empty or contain 1 element. When we do this, the result is $2 * 3 * 3 = 18$ test templates. At that stage, we decide not to partition the templates any further and we stop the process with these 18 leaf templates.

Applying the same type-based strategy to the other operations results in a total of 60 test templates for all the operations (all the other operations have fewer leaf templates than *lockRequestedByThread* because constraints on the valid input space of these operations restrict the number of possible combinations).

Instance and oracle templates. Consider the following leaf test template for *lockRequestedByThread*:

$$TT_{lockRequestedByThread} \mathrel{\widehat{=}}$$
$$\left[\, VIS_{lockRequestedByThread} \mid \right.$$
$$\left. \#isLockedBy = 1 \wedge \#isBlocking = 1 \wedge \#hasAskedToWait = 1 \,\right]$$

An instance template can be defined for this test template by instantiating the state variables and input:

$$IT_{lockRequestedByThread} \mathrel{\widehat{=}}$$
$$\left[\, isLockedBy : OneThread;\ isBlocking, hasAskedToWait : \mathbb{P}\ Thread; \right.$$
$$t? : Thread \mid t? = t_1 \wedge$$
$$\left. isLockedBy = \{t_2\} \wedge isBlocking = \{t_3\} \wedge hasAskedToWait = \{t_4\} \,\right]$$

where t_1, t_2, t_3 and t_4 are all distinct threads in *Thread*.

The oracle/output template for this instance template is:

$$OT_{lockRequestedByThread} \mathrel{\widehat{=}}$$
$$\left[\, isLockedBy' : OneThread;\ isBlocking', hasAskedToWait' : \mathbb{P}\ Thread \mid \right.$$
$$\left. isLockedBy' = \{t_2\} \wedge isBlocking' = \{t_1, t_3\} \wedge hasAskedToWait' = \{t_4\} \,\right]$$

Instance and oracle templates can easily be generated for the other 59 test templates as well.
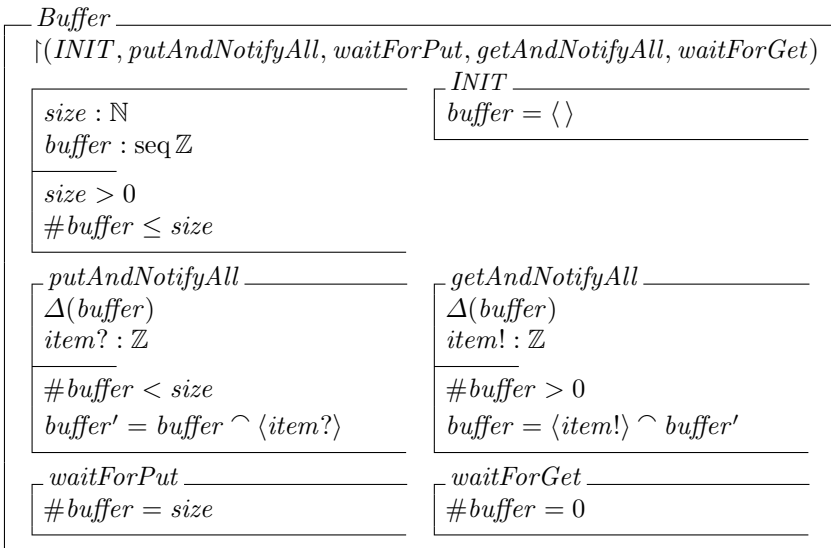
4   Concurrent Application Test Cases

4.1   Application Specification

Our approach to the specification of the application is to describe the different behaviours corresponding to the different thread paths through the synchronized object. This captures two important aspects of the application: (1) the effects on the application-specific variables, and (2) the conditions under which synchronisation occurs. The first aspect relates to the functional behaviour of the application that does not directly relate to its concurrent behaviour. That is, the effect on application-specific variables. The second aspect relates purely to the concurrent behaviour and captures the synchronisation policy of the application, i.e. the conditions under which the synchronisation primitives offered by the underlying concurrency mechanism should be invoked.

4.2   Example: Buffer

Our approach is illustrated by the following example specification of the Producer-Consumer problem. An Object-Z specification of the application-specific viewpoint of the finite buffer component presented in Figure 1 is now presented. The class provides the operations *putAndNotifyAll*, *waitForPut*, *getAndNotifyAll* and *waitForGet*. The internal mechanism of the buffer is modeled by a sequence. The *putAndNotifyAll* (and *getAndNotifyAll*) operation specifies the behaviour when a space for a resource (or a resource) is available in the buffer. These schemas also capture the conditions under which a notifyAll will be invoked. The *waitForPut* (and *waitForGet*) operation specifies the conditions in which a Producer (or Consumer) will wait until a put (or a get) is possible.

Buffer

$\upharpoonright(INIT, putAndNotifyAll, waitForPut, getAndNotifyAll, waitForGet)$

INIT
$buffer = \langle \rangle$

$size : \mathbb{N}$
$buffer : \text{seq}\, \mathbb{Z}$

$size > 0$
$\# buffer \leq size$

putAndNotifyAll
$\Delta(buffer)$
$item? : \mathbb{Z}$

$\# buffer < size$
$buffer' = buffer \frown \langle item? \rangle$

getAndNotifyAll
$\Delta(buffer)$
$item! : \mathbb{Z}$

$\# buffer > 0$
$buffer = \langle item! \rangle \frown buffer'$

waitForPut
$\# buffer = size$

waitForGet
$\# buffer = 0$

4.3   Buffer TTH

Test templates may be generated for the *Buffer* class by employing a type-based testing strategy. Variable *size* is examined by applying boundary analysis, e.g. the minimum size is 1 and we pick another "middle" size of 3. The specific values of the inputs and outputs may be left out of the test templates because the buffer is data-independent. Two examples follow from the 11 basic templates generated this way for the *Buffer* class.

$$TT_{putAndNotifyAll.1} \mathrel{\widehat{=}} \big[\, buffer : \mathrm{seq}\,\mathbb{Z};\ size : \mathbb{N} \mid \#buffer = 0 \wedge size = 1 \,\big]$$
$$TT_{waitForPut.1} \mathrel{\widehat{=}} \big[\, buffer : \mathrm{seq}\,\mathbb{Z};\ size : \mathbb{N} \mid \#buffer = 1 \wedge size = 1 \,\big]$$

$TT_{putAndNotifyAll.1}$ describes a case where a *put* operation should succeed and $TT_{waitForPut.1}$ describes a case where a *put* operation should wait.

Test instances and sequences. Test instances are generated from the templates by choosing appropriate values. Test oracles are created also. To enable test execution, test sequences must be created from the test instances. Test sequences have been generated automatically by bounded model checking using NuSMV [6]. Very briefly, we negate the test condition and get the model checker to produce the counter-example.

In practice, test sequences are comprised of calls to the Java component interface by different producer and consumer threads. Here, we use the interface offered by the Object-Z class. Sequences of executable method calls may be calculated from the operation sequences by taking into account the underlying JVM. (Some interactions may be infeasible in some versions of the JVM.) Example test sequences exercising test instances corresponding to the templates $TT_{putAndNotifyAll.1}$ and $TT_{waitForPut.1}$ follow.

$$(INIT(size = 1);\ putAndNotifyAll(item? = 1))$$
$$(INIT(size = 1);\ putAndNotifyAll(item? = 1);\ waitForPut)$$

Each test sequence begins with an *INIT* annotated with the size of the buffer (mimicking the Java constructor). The middle part of the sequence (which is empty for the first example) establishes the precondition of the desired test template. The last operation denotes a call to the operation in which the desired test template will be exercised.

While it is clear that checking the results of the execution of the test sequences against the oracles will verify the behaviour of the buffer with respect to the availability of the resource, the test sequences do not verify that the synchronisation mechanisms are called correctly. That is, we do not know whether notifyAll has been called rather than notify, or whether wait has been called when the *waitForPut*/*waitForGet* conditions occur. Verification of the suspension behaviour of the threads with respect to calls to wait and notifyAll is the topic of the next section.

5   A Concurrent Viewpoint of the Application

The complete specification of the concurrent behaviour of an application such as the buffer is formed by combining the application-specific behaviour as embodied in the *Buffer* class specified in the last section with the underlying concurrency model as embodied in the *Object* class specified in Section 2. It turns out that the complete specification of the behaviour involves not just application-specific detail but all of the detail of the synchronisation mechanism, and that test templates generated from the complete specification test the complete system, including the underlying synchronisation mechanism.

We will use the buffer example to first illustrate the problems inherent in testing the combination of application and synchronisation, and then see how to hide the internal mechanism of the *Object* class to produce test templates that focus solely on the application itself.

5.1   Buffer Object

The *Buffer* class and the *Object* class are combined to form the *BufferObject* class. It provides five (visible) operations.

$$
\begin{array}{|l}
\underline{\mathit{BufferObject}} \\[4pt]
\lceil(\mathit{lockRequestedByThread}, \mathit{putAndNotifyAll}, \mathit{waitForPut}, \\
\quad \mathit{getAndNotifyAll}, \mathit{waitForGet}) \\[6pt]
\mathit{Buffer} \\[6pt]
\mathit{Object} \\[6pt]
\mathit{putAndNotifyAll} \ \widehat{=} \\
\quad \mathit{giveLockToThread} \ \overset{\circ}{9} \ \mathit{notifyAllThreads} \ \overset{\circ}{9} \ \mathit{lockReleasedByThread} \\[6pt]
\mathit{waitForPut} \ \widehat{=} \ \mathit{giveLockToThread} \ \overset{\circ}{9} \ \mathit{askThreadToWait} \\[6pt]
\mathit{getAndNotifyAll} \ \widehat{=} \\
\quad \mathit{giveLockToThread} \ \overset{\circ}{9} \ \mathit{notifyAllThreads} \ \overset{\circ}{9} \ \mathit{lockReleasedByThread} \\[6pt]
\mathit{waitForGet} \ \widehat{=} \ \mathit{giveLockToThread} \ \overset{\circ}{9} \ \mathit{askThreadToWait} \\
\end{array}
$$

The first operation, *lockRequestedByThread*, is inherited unchanged from the *Object* class. This operation corresponds to a thread requesting a lock for entry into a synchronised block. The operation *putAndNotifyAll* specifies the behaviour when a Producer thread successfully puts some item into the buffer. The specification of this operation makes direct use of the Object-Z inheritance mechanism. To be specific, as the definition of *putAndNotifyAll*, namely,

$$
\mathit{putAndNotifyAll} \ \widehat{=} \\
\quad \mathit{giveLockToThread} \ \overset{\circ}{9} \ \mathit{notifyAllThreads} \ \overset{\circ}{9} \ \mathit{lockReleasedByThread}
$$

defined in the class *BufferObject* has the same name as an operation inherited from the class *Buffer*, it is conjoined with this inherited operation. The overall

result is that the specification of the operation *putAndNotifyAll* in the class *BufferObject* is equivalent to

$$putAndNotifyAll \;\widehat{=}\; putAndNotifyAll_{Buffer}$$
$$\wedge$$
$$(\,giveLockToThread_{Object}$$
$$\fatsemi\; notifyAllThreads_{Object}$$
$$\fatsemi\; lockReleasedByThread_{Object})$$

where subscripts are used in this expression simply to indicate the class from which the specific operation is inherited. The overall effect of the *putAndNotifyAll* operation is that, a thread, having already requested entry into the synchronised block (by way of membership of the set *isBlocking*), receives the lock, notifies all waiting threads, releases the lock, and at the same time achieves the effect of the *putAndNotifyAll* operation specified in the *Buffer* class. This style of specification, used for each of the other operations in the *BufferObject* class, emphasises the concurrent aspects of the operation and helps delineate the synchronisation mechanism of *Object*.

The operation *waitForPut* specifies the behaviour when a put is not possible and the thread has to wait. Upon receiving the lock, the thread finds that the condition for putting an item, as described by the *waitForPut* operation inherited from *Buffer*, does not hold and hence the thread waits.

The operations *getAndNotifyAll* and *waitForGet* are similar.

5.2   Test Case Selection

Applying the Test Template Framework to this class is complicated by the use of sequential composition to specify the combined operations. However, a procedure [18] exists for creating the test templates and oracles of Object-Z operations formed by combining other operations with conjunction, sequential composition, and parallel composition, out of the test templates and oracles of the component operations. In [18], Periyasamy and Alagar consider compositions of operations from a single class and without inheritance. Single inheritance has been considered elsewhere [17]; however, our buffer-object example inherits from multiple classes.

Multiple inheritance requires that the test templates for the inherited operations are promoted to the complete inherited state. Strategies for further developing the inherited templates should be carefully chosen to fit the design of the application. This approach is demonstrated on the *lockRequestedByThread* and *putAndNotifyAll* operations below.

Example: *lockRequestedByThread*. This operation is inherited unchanged from *Object* and becomes an operation of the *BufferObject* class. However, the state of the *BufferObject* class consists of the state of the *Object* class merged with the state of the *Buffer* class. The approach presented in [18] for building test templates from sub-components is to start with the union of the test templates

of the sub-components, promote the test templates to the combined state-space and then apply further test strategies to expand the promoted result. For instance, to generate the test templates for the promoted *lockRequestedByThread* operation, one has to conjoin each test template in the test template hierarchy for *lockRequestedByThread* with a schema which describes what happens to the *Buffer* state during the *lockRequestedByThread* operation. However, the *lockRequestedByThread* operation does not change the state of the *Buffer* component. This results in a set of test templates similar to the following.

$$TT_{lockRequestedByThread.i} \mathrel{\widehat{=}} \Xi Buffer.State \wedge Object.TT_{lockRequestedByThread.i}$$

Following this, one should apply test strategies to extend the test hierarchy further. A naive testing strategy for expanding this test hierarchy is to apply a type-based testing strategy to the promoted template.

If a 0-1-many testing strategy is applied to each promoted test template then the result will be $5 * 18 = 90$ test templates! However, these test templates are all re-testing the *lockRequestedByThread* operation in the presence of the application-specific inputs. This is testing two aspects.

1. It is re-testing *lockRequestedByThread*; exactly what we want to avoid, and
2. because the other operations, *putAndNotifyAll* etc. all rely on the given thread having already attempted entry to the synchronised block (as modelled by *lockRequestedByThread*) by way of the thread being in the set *isBlocking* (inherited from class *Object*), this is testing that a request for mutually exclusive access preceeds every other operation.

The first aspect should definitely be avoided, and more importantly, it is pointless to test the *lockRequestedByThread* for every application-specific input. However, by not testing *lockRequestedByThread*, there is a risk that exclusive access to the Object is not being verified.

In practice, it is impossible to test the correct use of *lockRequestedByThread* by black-box testing alone because the JVM manages the granting of locks by hidden internal operations. In light of this, the most sensible strategy for the black-box tester is to ignore the *lockRequestedByThread* operation[1].

Example: *putAndNotifyAll***.** Applying the approach outlined above, the set of base test templates for *putAndNotifyAll* is the union of the promoted test templates of *putAndNotifyAll* from *Buffer* and the promoted test templates corresponding to the sequence of synchronisation operations from *Object*.

The test templates for the sequence may be generated using the procedure outlined in [18]. However, the promotion of the operations suffers from the same problems as described above. In addition, the sequence of operations from *giveLockToThread* to *lockReleasedByThread* of *Object* completely hides the granting of the lock (*isLockedBy* equals the empty set at the start and at the

---

[1] In practice, code inspection is a more effective way to check for the correct use of synchronised blocks and methods.

end). Furthermore, as is the case for *lockRequestedByThread*, the state of the set of blocked threads *isBlocking* is completely hidden by the JVM.

The problems illustrated above demonstrate the infeasability of testing the *Buffer* with the Test Template Hierarchy developed from the combination of *Buffer* and *Object* classes. This leads us to the conclusion that this deep combination of *Buffer* and *Object* is not appropriate for producing black-box tests and motivates the more abstract model of the *Object* now presented.

5.3   Restricted Object Viewpoint

We now consider a restricted viewpoint of *Object* that captures the use of synchronisation by the application but that does not retest the underlying mechanism. We observe that the application controls the membership of the set *hasAskedToWait* by use of *wait* and *notifyAll* but that the JVM controls the blocking of threads and the granting of locks by *giveLockToThread*. In addition, the application class does not specify a behaviour for the *lockRequestedByThread* operation because the effect of the associated entry into a synchronised block or method is completely hidden. It is the "use" of an object that forms the basis of our restricted viewpoint. Variable and operation hiding is used to restrict the test cases generated for class *Object*. The locked thread represented by variable *isLockedBy* and the related operation *lockRequestedByThread* are hidden as well as the blocking set represented by *isBlockedBy* and the related *giveLockToThread* and *lockReleasedByThread* operations. The class *UseObject* defines the resulting class and is expanded below.

$$UseObject \; \widehat{=} \; Object \setminus (isLockedBy, isBlocking, lockRequestedByThread,$$
$$giveLockToThread, lockReleasedByThread)$$

UseObject
$\upharpoonright (INIT, askThreadToWait, notifyThread, notifyAllThreads)$

$hasAskedToWait : \mathbb{P} \; Thread$	**INIT** $hasAskedToWait = \varnothing$

askThreadToWait
$\Delta(hasAskedToWait)$
$t! : Thread$
───
$hasAskedToWait' =$
  $hasAskedToWait \cup \{t!\}$

notifyAllThreads
$\Delta(hasAskedToWait)$
$st! : \mathbb{P} \; Thread$
───
$st! = hasAskedToWait$
$hasAskedToWait' = \varnothing$

notifyThread
$\Delta(hasAskedToWait)$
$t! : OneThread$
───
$t! \subseteq hasAskedToWait$
$\#hasAskedToWait > 0 \Rightarrow \#t! = 1$
$hasAskedToWait' = hasAskedToWait \setminus t!$

UseObject Test Cases. The test templates resulting from applying the TTF to the *UseObject* class need only consider 0, 1, and many waiting threads. This gives 3 test templates for testing the correct application of each of the synchronisation mechanisms.

5.4 Application with UseObject

The application can be re-specified by the following.

┌─ *BufferUseObject* ──────────────────────────────
│ $\lceil(putAndNotifyAll, waitForPut, getAndNotifyAll, waitForGet)$
│
│ *Buffer*
│
│ *UseObject*
│
│ $putAndNotifyAll \mathrel{\widehat{=}} notifyAllThreads$
│
│ $waitForPut \mathrel{\widehat{=}} askThreadToWait$
│
│ $getAndNotifyAll \mathrel{\widehat{=}} notifyAllThreads$
│
│ $waitForGet \mathrel{\widehat{=}} askThreadToWait$
└──

By including the definition $putAndNotifyAll \mathrel{\widehat{=}} notifyAllThreads$ explicitly in the class *BufferUseObject*, we are associating the name *putAndNotifyAll* with the operation *notifyAllThreads* inherited from *UseObject*. This ensures that this operation is conjoined with the operation *putAndNotifyAll* inherited from *Buffer*. The other three visible operations in the class *BufferUseObject* are defined similarly.

BufferUseObject Test Cases. The base test templates for the combination are the union of the test templates for *Buffer* and the test templates for *UseObject*. As illustrated below, a common strategy for developing the test hierarchy further is to consider the different types of threads.

6 Strategy

This section summarises our derived strategy for testing concurrent components.

1. Specify the application-specific viewpoint. Define operations that cover all synchronisation paths through the monitor. That is, paths that start when a lock is granted (resulting either from the initial entry or from being notified after waiting) and end when the lock is released (either from exiting the synchronised block or from waiting). Operations should differentiate between paths that use notify, notifyAll, or use no notification.
2. Partition the application-specific viewpoint operations into a base test template hierarchy.

3. Combine the test template hierarchy of the application-specific model with that of the restricted concurrency model to introduce thread suspension behavior.
4. Use the number and type of waiting threads to further develop the resultant test template hierarchy.

In step 1 we have used the underlying thread mechanism provided by the *Object* class to decide the synchronisation points that should be covered by the operations. In other work [13] we have used Concurrency Flow Graphs to produce the test conditions. In future work we will look at how these two approaches can be combined.

Step 2 is a standard application of the Test Template Framework to the viewpoint specification produced in Step 1.

Step 3 requires some ingenuity on the part of the tester to decide strategies that take advantage of the component design to avoid re-testing the underlying Java mechanism and to focus on the synchronisation under control of the application.

Step 4 is possible because of the introduction of the threads themselves in step 3. Test partitioning based on the number and type of suspended threads is standard for this type of application because it shows whether particular classes of thread (such as Producers or Consumers) are starved because of inappropriate wait conditions or notification.

As a further demonstration of this strategy we next apply it to the Readers-Writers problem.

6.1   Case Study: Readers-Writers

The Readers-Writers problem involves a shared resource which is read by reader threads and written to by writer threads. To prevent non-interference, individual writers must be given exclusive access to the resource, locking out other writers and readers. However, reading does not result in interference and so multiple readers may access the resource concurrently. A monitor is used to control access to the resource. Our approach is built on that presented in standard concurrency textbooks [2,16].
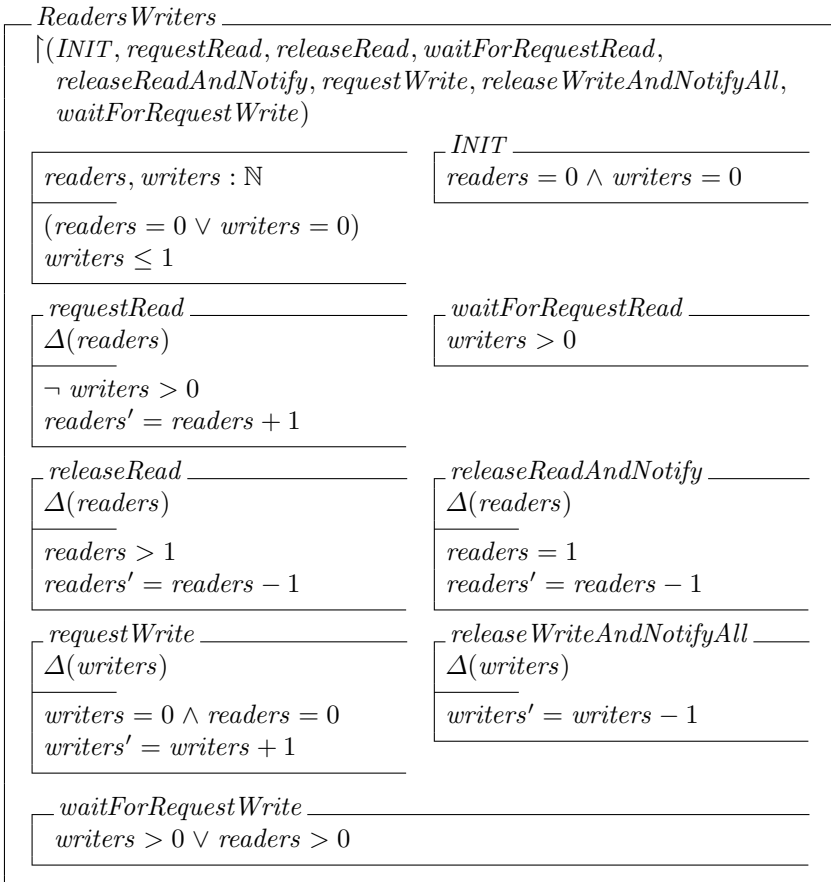
Step 1. We specify the application-specific behaviour of the Readers-Writers monitor by considering all paths through the synchronisation points. Object-Z class *ReadersWriters* (presented below) specifies operations corresponding to the different ways in which a thread may progress through the monitor.

The state of the monitor involves two counters corresponding to the number of *readers* and *writers* concurrently accessing the resource. The state invariant captures the desired *monitor invariant*: the resource is accessed either by concurrent readers or writers but never by both, and the number of concurrent writers is never greater than one. Initially there are no readers or writers.

The operations capture the application-specific aspects only. The concurrent behaviour is added in Step 3.

A read request will succeed immediately if there are no writers. This is captured by operation *requestRead*. A read request will be delayed (the thread waits) if there is currently a writer accessing the resource (*waitForRequestRead*). Once finished reading, a reader releases the resource. There are two cases, if the thread is not the last reader, i.e., *readers* > 1 before the release, then the readers just stops reading as per *releaseRead*. However, if the reader is the last thread (*readers* = 1) then the reader notifies a waiting writer[2], as specified in *releaseReadAndNotify*.

A request for write access will succeed immediately if the number of readers and writers both equal 0 (*requestWrite*). A request for write access will wait otherwise (*waitForRequestWrite*). When releasing write access, a thread always notifies all other waiting threads (*releaseWriteAndNotifyAll*).

ReadersWriters
⌈(*INIT*, *requestRead*, *releaseRead*, *waitForRequestRead*,
  releaseReadAndNotify, *requestWrite*, *releaseWriteAndNotifyAll*,
  waitForRequestWrite)

INIT
$readers = 0 \wedge writers = 0$

$readers, writers : \mathbb{N}$

$(readers = 0 \vee writers = 0)$
$writers \leq 1$

requestRead
$\Delta(readers)$

$\neg\, writers > 0$
$readers' = readers + 1$

waitForRequestRead
$writers > 0$

releaseRead
$\Delta(readers)$

$readers > 1$
$readers' = readers - 1$

releaseReadAndNotify
$\Delta(readers)$

$readers = 1$
$readers' = readers - 1$

requestWrite
$\Delta(writers)$

$writers = 0 \wedge readers = 0$
$writers' = writers + 1$

releaseWriteAndNotifyAll
$\Delta(writers)$

$writers' = writers - 1$

waitForRequestWrite
$writers > 0 \vee readers > 0$

---

[2] It notifies any waiting thread but only writing threads will wait for a reader to release.

6.2   Step 2

Test templates are now generated for the application-specific viewpoint.

Many of the test templates correspond to the valid input space because the preconditions are so simple. In the cases of *requestRead* and *releaseRead* we apply a type-based strategy on the number of readers (0-1-many), and in the case of *waitForRequestWrite* we apply domain partitioning to the top-level disjunction and a type-based strategy; 11 base test templates are produced in this way.

6.3   Step 3

The application-specific viewpoint is now combined with the concurrency viewpoint capturing thread suspension behaviour. The class *ReadersWritersObject* describes the combination. It extends all *ReadersWriters* operations with the appropriate concurrent behaviour. As operations *requestRead*, *releaseRead*, and *requestWrite* all succeed immediately (without suspension) and do not notify any other threads, they do not need to be combined with any synchronisation operations. Operations *waitForRequestRead* and *waitForRequestWrite* are combined with *askThreadToWait* because they capture the waiting behaviour for a read or write request. Operation *releaseReadAndNotify* is combined with *notifyThread* because it captures the case when the last reading thread releases the resource and must notify a waiting writer (if one exists). Similarly, *releaseWriteAndNotifyAll* is combined with *notifyAllThreads* because all waiting threads must be notified when a writer releases the resource.

$$
\begin{array}{|l}
\hline
\,\_\_\, ReadersWritersObject \,_____ \\
\upharpoonright(requestRead, waitForRequestRead, releaseRead, releaseReadAndNotify, \\
\quad requestWrite, waitForRequestWrite, releaseWriteAndNotifyAll) \\[4pt]
ReadersWriters \\[2pt]
UseObject \\[4pt]
waitForRequestRead \mathrel{\widehat{=}} askThreadToWait \\
releaseReadAndNotify \mathrel{\widehat{=}} notifyThread \\
waitForRequestWrite \mathrel{\widehat{=}} askThreadToWait \\
releaseWriteAndNotifyAll \mathrel{\widehat{=}} notifyAllThreads \\
\hline
\end{array}
$$

Combined TTH. Test templates for the combined operations are constructed from the union of the test templates of the sub-operations. In the case of *requestRead* and other operations inherited directly from *ReaderWriters*, the test templates are just the ones inherited from the *ReadersWriters* test template hierarchy. In the case of the combined *waitForRequestRead*, *releaseReadAndNotify*, *waitForRequestWrite*, and *releaseWriteAndNotifyAll* operations, the combined test templates result from the union of test templates from the *ReadersWriters* class and those from the *UseObject* class.

6.4   Step 4

To further develop the test templates for the operations *waitForRequestRead*, *releaseReadAndNotify*, *waitForRequestWrite*, *releaseWriteAndNotifyAll*, we consider 0, 1, or many reader and writer threads waiting. This allows us to test, for instance, that all waiting threads are notified when *releaseWrite* is called.

7   Conclusion

While others have applied the Test Template Framework to interactive systems, in this paper we apply it to concurrent components. We have focused on the Java concurrency model but our approach could be generalised to other concurrency models such as protected Ada Objects.

Our approach has been to separate out the application and underlying concurrency mechanism into separate viewpoints and then to develop test hierarchies for them separately. We have then combined the test hierarchies for different viewpoints taking into account the designed isolation of the concurrency mechanism. This demonstrates a new approach to test template generation. In doing so we have had to deal with multiple inheritance, a previously untreated aspect of the application of the test template framework to object-oriented specifications.

References

1. Bernhard K. Aichernig. Test-case calculation through abstraction. In *Proceedings of Formal Methods Europe 2001, FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 571–589. Springer-Verlag, 2001.
2. G. Andrews. *Concurrent Programming: Principles and Practice*. Addison Wesley, 1991.
3. H. Bowman, M.W.A. Steen, E.A. Boiten, and J. Derrick. A formal framework for viewpoint consistency (full version). Computing Laboratory Technical Report 22-99, University of Kent at Canterbury, Canterbury, Kent, CT2 7NZ, December 1999.
4. Marius C. Bujorianu, Savi Maharaj, and Manuela Bujorianu. Towards a formalization of viewpoints testing. In Robert M. Hierons and Thierry Jéron, editors, *Formal Approaches To Testing of Software 2002 (FATES'02)*, Research Report, 35042 Rennes, France, August 2002. INRIA. A satellite workshop of CONCUR'02.
5. David Carrington, Ian MacColl, Jason McDonald, Leesa Murray, and Paul Strooper. From Object-Z Specifications to ClassBench Test Suites. *Software Testing, Verification and Reliability*, 10(2):111–137, 2000.

6. Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

7. R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z*. Cornerstones of Computing. Macmillan Press Limited, UK, 2000.

8. Roger Duke, Luke Wildman, and Brad Long. Modelling Java Concurrency with Object-Z. In A. Cerone and P. Lindsay, editors, *Software Engineering and Formal Methods (SEFM'03)*, pages 173–181. IEEE Computer Society Press, 2003.

9. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000. Also online at http://java.sun.com/docs/books/jls/index.html as at Sep 2002.

10. Daniel Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–389, 1995.

11. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999.

12. Brad Long, Dan Hoffman, and Paul Strooper. Tool support for testing concurrent Java components. *IEEE Transactions of Software Engineering*, 29(6):555–566, June 2003.

13. Brad Long and Paul Strooper. A Classification of Concurrency Failures in Java Components. In *Proceedings of the 1st International Workshop on Parallel and Distributed Systems: Testing and Debugging*, April 2003.

14. I. MacColl and D. Carrington. Testing matis: A case study on specification-based testing of interactive systems. In *Formal Aspects of HCI (FAHCI98)*, pages 57–69, 1998.

15. Ian Dugald MacColl. *Specification-Based Testing of Interactive systems*. PhD thesis, Information Technology and Electrical Engineering, The University of Queensland, Feb 2003.

16. J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley & Sons, 1999.

17. L. Murray, D. Carrington, I. MacColl, and P. Strooper. Extending test templates with inheritance. In Paul A. Bailes, editor, *Proceedings of the Australian Software Engineering Conference ASWEC'97*, pages 80–87. IEEE Computer Society, September 1997.

18. K. Periyasamy and V.S. Alagar. A rigorous method for test templates generation from object-oriented specifications. *Software Testing, Verification and Reliability*, 11:3–37, 2001.

19. Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on software Engineering*, 22(11):777–793, November 1996.