A UML Approach to the Generation of Test Sequences for Java-based Concurrent Systems

Soon-Kyeong Kim, Luke Wildman and Roger Duke
School of Information Technology and Electrical Engineering
The University of Queensland, St Lucia, Australia
Email: {soon, luke, rduke}@itee.uq.edu.au

Abstract

Starting with a UML specification that captures the underlying functionality of some given Java-based concurrent system, we describe a systematic way to construct, from this specification, test sequences for validating an implementation of the system. The approach is to first extend the specification to create UML state machines that directly address those aspects of the system we wish to test. To be specific, the extended UML state machines can capture state information about the number of waiting threads or the number of threads blocked on a given object. Using the SAL model checker we can generate from the extended UML state machines sequences that cover all the various possibilities of events and states. These sequences can then be directly transformed into test sequences suitable for input into a testing tool such as ConAn. As an illustration, the methodology is applied to generate sequences for testing a Java implementation of the producer-consumer system.

1. Introduction and motivation

Testing is an essential part of validating any concurrent system. In order to adequately test a concurrent system we need to have a precise specification of the system itself, a precise specification of the model of concurrency adopted by the underlying implementation programming language, and a sound strategy for generating test sequences that fully exercise the relevant aspects of the system.

In this paper we use UML [13] to develop a state machine model that captures the underlying functionality of Java concurrency. We then extend this model to explicitly include those aspects of the concurrent system we wish to test. For example, in this paper we extend the state machine model with state information about whether none, one or many threads are blocked or waiting on an object. From this extended model we can then systematically generate test sequences by using a model checker such as SAL [11] that cover all the relevant possibilities of events and states. The test sequences produced can be executed on the system using a testing tool such as ConAn [9]. As an illustration we apply the technique to a producer-consumer system.

In previous work we have developed an approach to the problem of testing concurrent systems [5, 14]. The basis for this approach has been to first develop formal models capturing both the system itself and the underlying Java concurrency mechanism, and then from these models create appropriate test sequences. Although this approach is sound, and is indeed the basis of the approach taken in this paper, a significant problem is that it is not easy to systematically generate test sequences from these models. Probably the most important cause of this difficulty is, to put it simply, complexity. Java concurrency is complicated, and any attempt to formally capture this complexity inevitably leads to a specification which is often quite tricky and not easy to work with. In this paper we take an approach that uses UML class diagrams and state machines to model concurrency as it gives us the best of both views; the notation is sufficiently formal to capture the precision we need, but sufficiently informal to enable us to restrict attention to the essentials. In addition, the graphical nature of UML enhances the readability and hence applicability of the models.

Our approach is to start with a UML model of the underlying Java concurrency mechanism, but to then extend this model to capture explicitly features of the model that are directly relevant to the generation of test sequences. By using UML state machines as the underlying modelling formalism it is relatively easy to create viewpoints that abstract away from unneeded detail and focus just on those aspects of the concurrent system that are most relevant from a testing perspective. Not only are the resulting state machines models easier to comprehend than a full-blown formal specification, but test sequences can be generated from these models in a systematic way.

In other work [10] we have developed a method for testing concurrent Java components. Part of this

1530-0803/05 $20.00 © 2005 IEEE

method involves a testing strategy for selecting test sequences for ConAn. In particular this strategy varies the number and type of threads waiting on an object when a notification is generated, in order to test that wait conditions and notifications have been implemented correctly. This condition has been chosen as the focus of this paper; however the general approach may be applied for any set of test conditions.

While the majority of the UML based test generation strategies focus on system testing (see [1, 2] for example), there has been some work on test case generation for components. For instance a method for generating concrete test cases from abstract test sequences has been developed [3]. This approach assumes that abstract test sequences, in the form of (state, event) lists, have been prepared by the user to satisfy some test conditions. Our approach does not make this assumption; we generate the concrete test cases directly from the test conditions.

2. The underlying UML model of Java Concurrency

In this section, we briefly describe concurrency in Java. 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 integers. The *put* method is used to add an integer to the finite buffer. The calling thread waits if space is not available; if space is available, it notifies all waiting threads after it adds the integer. The *get* method retrieves an integer from the buffer. The calling thread waits if a resource is not available; if a resource is available, it notifies all other waiting threads after it removes the integer. 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 this paper, we develop a UML model of a class Object to capture the underlying Java concurrency mechanism consistent with that described within the Java Language Specification [6] and the Java Virtual Machine (JVM) Specification [8]. We shall assume a basic understanding of the Java synchronisation model, although in fact the issues will be briefly discussed here as part of the description of the UML state machine of a Java class named Object.

```
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);
   }

}
```

Figure 1. Java concurrency application

A basic understanding of UML is also assumed (see [7] for the UML state machine background).

Instances 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.

Figure 2 presents a simple class diagram showing the class Object and its relationships with the class Thread. The three associations between the two classes indicate that at most one thread can lock an object and there can be a set of threads either asked to wait or blocked on the object. Since any thread can play at most one of these relationship roles for any given object, the three associations are disjoint. The {disjoint} constraint denoting mutually disjoint associations is added to the associations.

We also assume that each class has private operations that maintain the list of objects associated with each of the association ends attached to the opposite class. For example, the class Object has add and remove operations (addToBlocking and removeFromBlocking) that ultimately maintain threads that are currently associated with the object via the blocking association role.

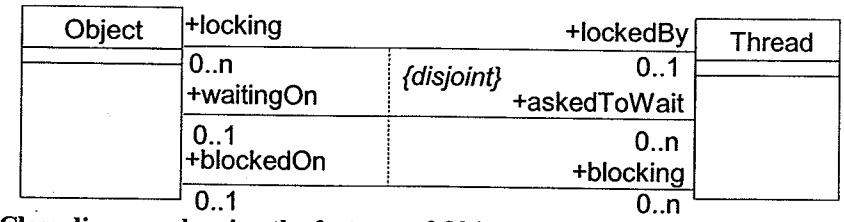

Figure 2. Class diagram showing the features of Objects and Threads, and their relationships

2.1. The class *Object*

A UML state machine describing the behaviour of the class Object is presented in Figure 3. The diagram has a top state (Object) that models a synchronized block of the object. This top state has two sub-states: *Locked* and *UnLocked* respectively models the state when the object is locked by a thread and when the object is not locked by any thread. When the object is in the *UnLocked* state, the association end lockedBy must be empty. In contrast, if the object is in the *Locked* state, the association end lockedBy must not be empty. Given these states, we describe the object behaviour in terms of transitions that specify what happens when the object receives requests (events) in the states.

Note that in the diagram, we describe guard conditions and transition specifications (e.g. action list, send event and send target) in a separate table to reduce the complexity of the diagram (See Table 1). Also note that we use OCL [12] to specify the guard conditions of the transitions. The actions are executed by the order defined in the table.

Consider now the state machine of the class *Object* in detail. The transition *lockRequest* specifies what happens when a thread requests access to the object. This captures the object's view of the situation when a Java thread seeks to enter a synchronized block of the object. The guard condition of the transition (G1) ensures that the thread in question cannot already be locking the object, cannot be blocked on the object, and cannot be waiting to be notified by the object. Since the object can respond to this request from any sub-state as long as the guard conditions are satisfied, the transition is attached to the top state (Object). As a result of this transition, the object adds the requesting thread into the list of blocking threads of the object (see the action of T1 in Table 1).

When the object is locked by a thread (in the Locked state), it can respond to several requests (events) as follows:

- a request to release the thread currently holding the lock on the object (see the transition caused by the event *lockRelease*). This captures the object's view of the situation when a Java thread completes the execution of a synchronized block of the object. The result of this transition is that the object is no longer locked (changing the state from Locked to UnLocked), the thread is removed from the lockedBy list (see the action of T2).
- a request to notify a thread currently waiting on the object (see the transition caused by the event *notify*). This captures the object's view of the situation when a 'notify' method is executed by some other thread currently accessing a synchronized block of the object. The result of this transition is that the object selects one thread from the currently waiting threads (askedToWait) and sends a notification to the thread (see a synchronization send event, notifiedBy of T3), and removes the thread from the askedToWait list and adds it into the blocking list (see the action of T3).
- a request to notify all threads currently waiting on the object (see the transition caused by the event *notifyAll*). This captures the object's view of the situation when a 'notifyAll' method is executed by some other thread currently accessing a synchronized block of the object. The result of this transition is that all the threads currently waiting on the object (askedToWait) are notified (see a synchronization send event, *notifiedBy* of T4), and the object removes them from the askedToWait list and adds them to the blocking list (see the action of T4)

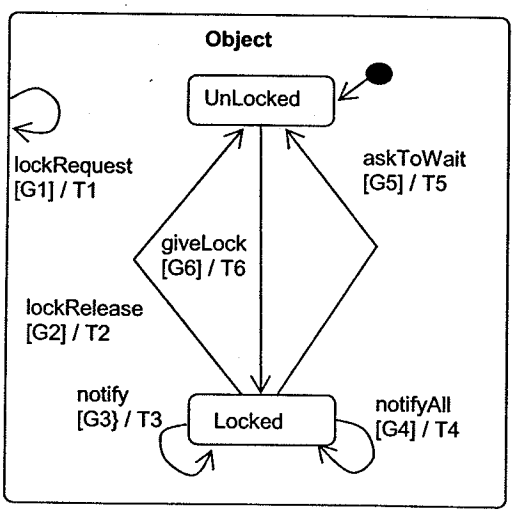

Figure 3. UML State Machine for Object

Event	Guard		Transition Specification			
				Action	Send event	Send target
lockRequest	G1	requestingThread -> not include (lockedBy) requestingThread -> not include (blocking) requestingThread -> not include (askedToWait)	T1	addToBlocking (requestingThread)		
lockRelease	G2	requestingThread -> include(lockedBy)	T2	removeFromLockedBy (requestingThread)		
notify	G3	selectedThread -> include(askedToWait) requestingThread -> include(lockedBy)	T3	removeFromAskedTo Wait(selectedThread), addToBlocking (selectedThread)	notifiedBy	self.askedToWait -> select (t | t = selectedThread)
notifyAll	G4	selectedThreads -> include(askedToWait) requestingThread -> include(lockedBy)	T4	removeFromAskedTo Wait(allWaitingThreads), addToBlocking (allWaitingThreads)	notifiedBy	self.askedToWait -> forAll (t)
askToWait	G5	requestingThread -> include(lockedBy)	T5	removeFromLockedBy (lockedByThread), addToAskedToWait (lockedByThread)	askedToWait	self.lockedBy -> forAll (t)
giveLock	G6	selectedThread -> include(blocking)	T6	removeFromBlocking (selectedThread), addToLockedBy (selectedThread)	getLockOn	selectedThread

Table 1. The transition specifications of the Object State Machine

- a request to wait to the thread currently holding the lock on the object (see the transition caused by the event *askToWait*). This captures the object's view of the situation when a Java thread executes a 'wait' method while accessing a synchronized block of the object. The result of this transition is that the object changes its state from *Locked* to *UnLocked* and the thread is removed from the lockedBy list and added into the askedToWait list (see the action of T5). Also a synchronization event *askedToWait* is sent to the thread (see the send event and send target of T5).

Similarly, when the object is not locked by any thread (is in the *UnLocked* state), it can also respond to the following request:

- a request to lock the object (see the transition caused by the event *giveLock*). The object selects a thread from among those currently blocked by the object and allows the thread to lock it. This captures the object's view of the situation when a Java thread is given access to a synchronized block of the object. The result of this transition is that the object is locked (changing the state from *UnLocked* to *Locked*), the thread is removed from the blocking list and added to the lockedBy list (see the action of T6), and sends a synchronization event *getLockOn* to the thread (see the send event and send target of T6).

2.2. The class *Thread*

In the class diagram in Figure 2, the class Thread has three relationships with the class Object: locking, blockedOn and waitingOn which denote respectively a set of locking objects, the object (if any) blocking the thread, and the object (if any) on which the thread is waiting for notification. We model the behaviour of a thread similar to the object behaviour. Due to space limitations, we do not present the model in the paper.

2.3 The extended UML test model

In the previous sections, we modelled objects and threads using the UML state machine to understand their general behaviour. In this section, we are interested in their behaviour under certain situations in order to develop a testing model of objects.

To do this, we extend the UML models developed in the previous sections by including testing aspects into the models. In detail, we define a separate composite state at the top level of the existing UML state machine. The composite state includes a set of states (interesting test states). These test states are defined based on a test case selection strategy. Given these test states, we can model an object's or thread's behaviour from a testing point of view. This extended testing model will also be used as a basis to derive test sequences described in Section 4.

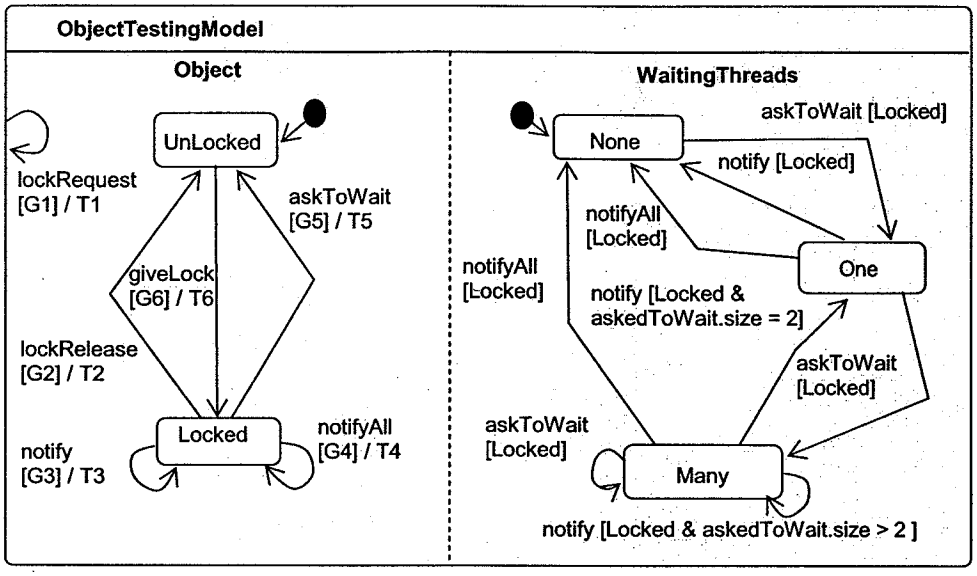

Figure 4. UML State Machine modelling multiple waiting threads on an object from a testing aspect

The diagram in Figure 4 is an extended UML model for objects. In the example, we are interested how the object will behave when there is no waiting thread, one waiting thread and many waiting threads. The transitions show what happens when the object receives certain events such as wait, notify and notifyAll when there is none, one, and many threads waiting on the object. Actual transitions in these testing states are triggered by the actions executed as a result of the transitions triggered by the same events in the original model (see the action list in Table 1).

The extended testing model synchronizes with the original model in terms of the guard conditions given for the transitions defined in the extended model. For example, the transitions defined in the *WaitingThreads* state can happen only when the object is locked i.e. when in the *Locked* state in the original model.

3. Applying the model to Producer-Consumer

Now that we have a precise model of Java concurrency we can apply this model to software systems such as the classical producer-consumer. It is straightforward to specify this problem in UML in a way that ignores concurrency issues. We begin by developing a class diagram including the classes *Producer*, *Consumer* and *Buffer* (see Figure 5). A ProducerConsumer system has a set of producers, consumers and buffers. The class Producer has an operation (*put*) with an output parameter named *item*. Similarly the class Consumer has an operation (*get*) with an input parameter named *item*. The class Buffer has an attribute *buffer* to store items in order and another attribute *max* recording the size of the buffer, and has two operations (*get* and *put*) to get and put an item from the buffer respectively. This class diagram models a sequential (non concurrent) consumer-producer system.

A UML state machine showing the behaviours of a buffer is presented in Figure 6. This state machine models a sequential buffer.

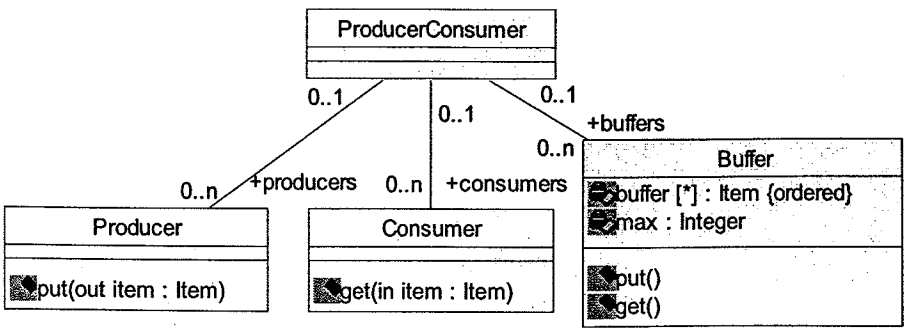

Figure 5. A sequential Producer-Consumer system

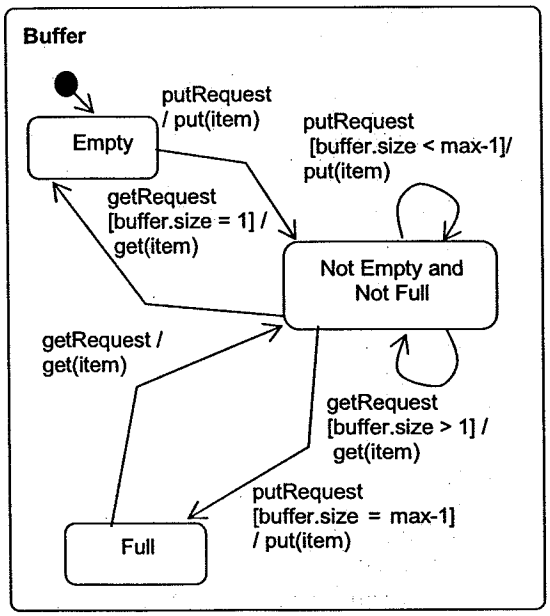

Figure 6. UML state machine modelling buffer

We now introduce concurrency in the producer-consumer problem. We can do this by reusing the earlier UML model of Java concurrency with the generalization relationship in UML. The class diagram in Figure 7 models a concurrent Producer-Consumer system. In the diagram, the classes Producer and Consumer inherit from the class Thread, and the class BufferMonitor inherits from both the class Object and the class Buffer.

By the semantics of generalization in UML which supports both subtyping and inheritance, the subclasses of the class Thread, Producer and Consumer inherit all the features of the class Thread including the relationships with the class Object. Similarly the class Buffer also inherits all features of the class Object. Although the UML semantics document [13] does not formally describe how to interpret the UML state machine in the context of inheritance, it is common to interpret it based on the general semantics of inheritance in object technologies [12]. That is, when a UML state machine is attached to a class to describe its behaviour, the behaviour must be preserved in its inheriting classes. This simply means that features in the state machine of the super class can be replaced, but not deleted, and new features can be added.

In Figure 8, we show a refined UML state machine for a concurrent buffer. The two UML state machines developed for the classes Object and Buffer are composed into a composite state. Several actions are added to the transitions in the *Buffer* state to synchronize with the *Object* state. For example, the transitions triggered by events requesting *get* or *put*, can happen only when the buffer is currently locked by the requesting thread (see the guard conditions in Table 2 extending Table 1), thus this constraint is added as a guard condition of the transitions. When a put or a get request is completed, the buffer notifies this to all threads waiting and also sends an event releaseLockOn to the currently locking thread to unlock itself consequently changing the buffer state from *Locked* to *UnLocked* (see T7 and T8 for example).

When the buffer is empty and there is a request for *get*, the buffer undergoes an internal event *askToWait* to ask the currently locking thread to wait (see G7 and T7). This is similar to when the buffer is full and there is a thread requesting to put an item into the buffer (see G14 and T14).

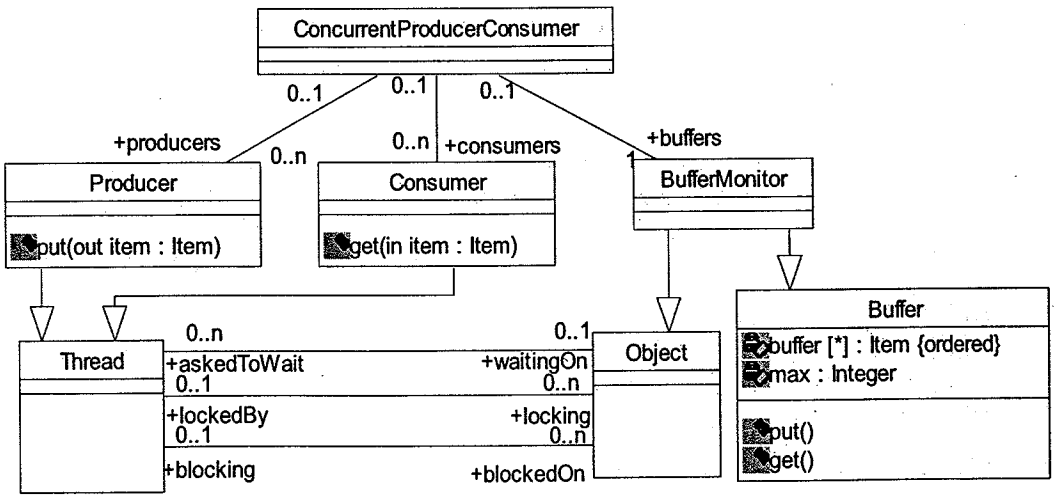

Figure 7. A concurrent Producer-Consumer system

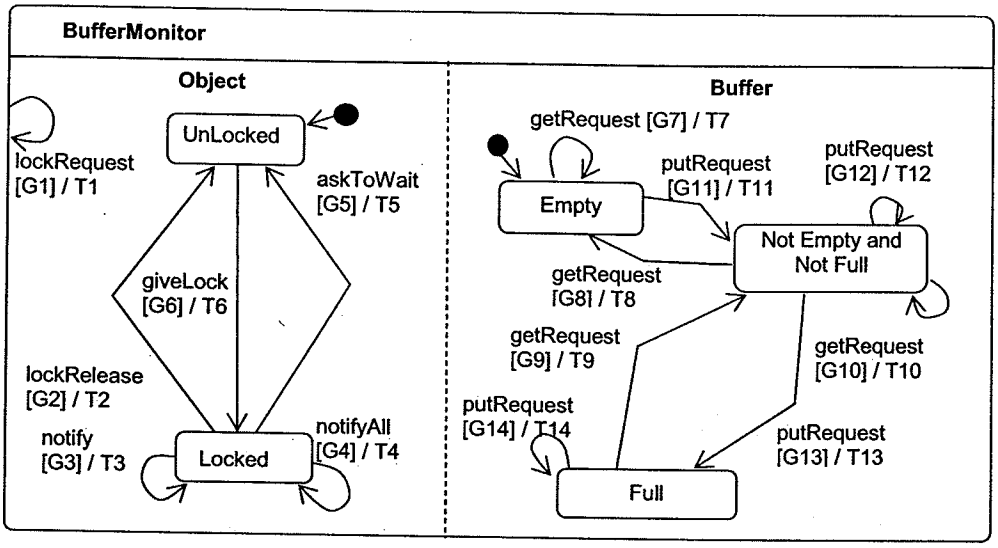

Figure 8. UML state machine modelling a concurrent buffer

Event	Guard		Transition Specification			
				Action	Send event	Send target
getRequest	G7	Locked and requestingThread -> include(lockedBy)	T7		askToWait	self
getRequest	G8	Locked, requestingThread -> include(lockedBy) and buffer.size = 1	T8	get(item)	notifyAll releaseLockOn	self lockingThread
getRequest	G9	Locked, requestingThread -> include(lockedBy)	T9	get(item)	notifyAll releaseLockOn	self lockingThread
getRequest	G10	Locked, requestingThread -> include(lockedBy) and buffer.size > 1	T10	get(item)	notifyAll releaseLockOn	self lockingThread
putRequest	G11	Locked, requestingThread -> include(lockedBy)	T11	put(item)	notifyAll releaseLockOn	self lockingThread
putRequest	G12	Locked, requestingThread -> include(lockedBy) and buffer.size < max-1	T12	put(item)	notifyAll releaseLockOn	self lockingThread
putRequest	G13	Locked, requestingThread -> include(lockedBy) and buffer.size = max -1	T13	put(item)	notifyAll releaseLockOn	self lockingThread
putRequest	G14	Locked and requestingThread -> include(lockedBy)	T14	put(item)	askToWait	self

Table 2. The transition specifications of the BufferMonitor State Machine

3.1 The extended test model of BufferMonitor

We now develop a test model for the concurrent buffer in the same way that a test model of objects was developed in Section 2.3. For a concurrent buffer, we are interested in how the concurrent buffer will behave when there is none, one and many producers. Similarly when there is none, one and many consumers. In Section 2.3, we modelled the object behaviour when there is none, one and many waiting threads by adding a test state to the original object model (see the state *WaitingThreads* in Figure 4). We capture the buffer behaviour a similar way. That is, we introduce a new test state for each of the waiting producers and consumers (see *WaitingProducers* and *WaitingConsumers* in Figure 9). Since producers and consumers are waiting threads, they are members of the askedToWait list of the buffer object. Thus, the *None* sub-state in the *WaitingProducers* state denotes a situation where no producer thread is waiting in the askedToWait list. Similarly the *None* sub-state in the waiting in the *WaitingConsumers* state denotes a situation where no consumer thread is waiting in the askedToWait list. Similar conditions hold in all the other sub-states defined in the two test states.

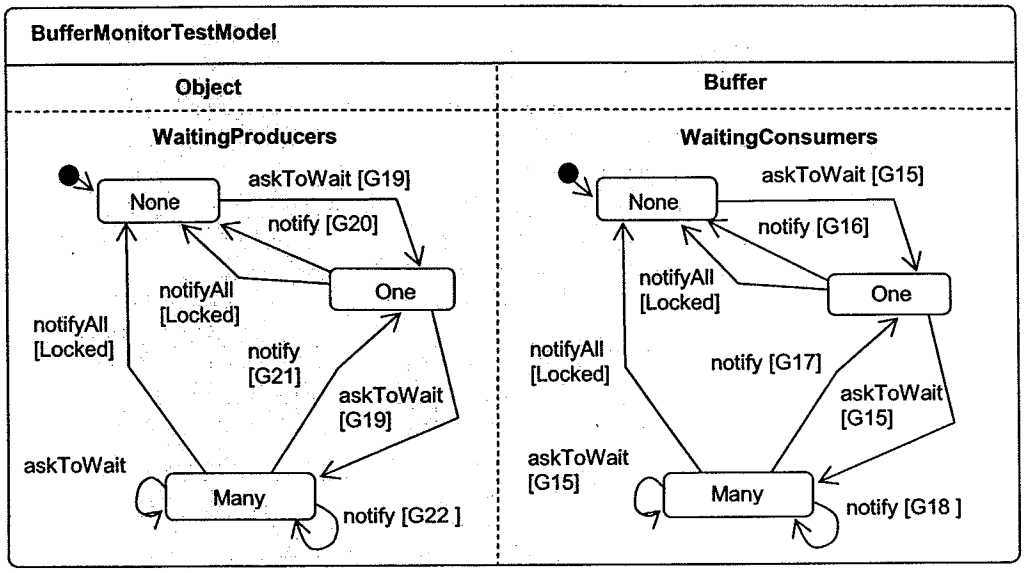

Figure 9. UML State Machine modelling multiple producers and consumers from a testing aspect

Event	Guard	
askToWait	G15	Locked, requestingThread -> include(lockedBy) and requestingThread -> instanceOf(Consumer)
notify	G16	Locked, selectedThread -> instanceOf(Consumer)
notify	G17	Locked, selectedThread -> instanceOf(Consumer) and self.askedToWait -> select (t | t -> instanceOf(Consumer)) -> size = 2
notify	G18	Locked, selectedThread -> instanceOf(Consumer) and self.askedToWait -> select (t | t -> instanceOf(Consumer)) -> size > 2
askToWait	G19	Locked, requestingThread -> include(lockedBy) and requestingThread -> instanceOf(Producer)
notify	G20	Locked, selectedThread -> instanceOf(Producer)
notify	G21	Locked, selectedThread -> instanceOf(Producer) and self.askedToWait -> select (t | t -> instanceOf(Producer)) -> size = 2
notify	G22	Locked, selectedThread -> instanceOf(Consumer) and self.askedToWait -> select (t | t -> instanceOf(Producer)) -> size > 2

Table 3. The transition specifications of the test states defined in the BufferMonitor test model

Guard conditions of the transitions defined in these test states are also expressed in terms of the askedToWait list and the type of threads in the list (see Table 3). Note that in the diagram, we suppress the states (*Object* and *Buffer*) modelling the original behaviour of buffer (see Figure 8) to reduce complexity.

4. Generating test sequences from the BufferMonitor model.

We now explain how test sequences may be generated from the extended State Machine models. A manual translation of the UML model to a SAL model is performed, this translation is systematic but still under development. The SAL model is then combined with an additional model that captures the particular testing strategy used by ConAn. The resultant model is then used to create counter-examples which may then be used to create test sequences for ConAn. The details of the ConAn testing strategy and the counter-example generation are now described.

4. 1 Testing in Conan

The ConAn testing tool uses a deterministic execution strategy to the test the component. The deterministic testing strategy uses a clock to control the execution of calls on the monitor in such a way that all threads currently blocked or waiting on the monitor at a particular time will be able to get the lock, finish executing (including making calls to notifyAll), and

return to waiting (if woken after notification and failing the wait condition). This 'run-to-completion' strategy allows the tester to check for correct use of the Java synchronized, wait, and notify statements by creating scripts that call monitor methods at given times, and check for termination of those calls at later times. In order to automatically create test sequences based upon the 'run-to-completion' strategy, the corresponding constraint is combined with the SAL translation of the BufferMonitor model.

4.2 Generating Test sequences

Most model checkers generate counter-examples to invalid propositions. A counter-example that can be used to construct a test sequence that reaches a desired test-condition (TC) can be generated by running the model checker on the Linear Temporal Logic (LTL) proposition *G (NOT TC)*. Test sequences for ConAn are created by calling methods and then checking when they terminate. A test-sequence for a particular test-condition is a sequence which reaches a state corresponding to that test condition and then continues until a certain exit state (*Exit*) in which all threads are terminated. This is stated in LTL as *G(NOT (TC AND F(Exit)))* where Exit is the state (*Blocking = empty* and *Waiting = empty*). If in addition the tester wants to observe the outputs produced by the terminating threads (for the purpose of checking the content of the buffer for instance), the Exit state may be extended with monitor conditions, e.g, (*Exit and buffer = empty*).

For example, a counter example that tests the condition *ManyProducersWaiting* and returns the monitor to the Exit state is produced by the following proposition.

```
th1:THEOREM system |-
      G (NOT (ManyProducersWaiting  AND
              F (Exit AND BuffefEmpty)))
```

4.3 Dealing with a non-deterministic JVM

The JVM specification is non-deterministic and may be implemented differently on different platforms. In particular, there is no guarantee that blocked threads will be served in first-come first-served order, nor that the waiting threads will be served in order. Our model make no such assumptions, however, in generating the counter example, choices made by the JVM are simulated by the model checker, and it may be the case that the model checker chooses a different thread to the particular implementation under test.

In order to test a particular platform it would be necessary to encode the pattern of choices made by that platform into our model. However, ConAn allows sets of possible termination times and outputs to be specified, providing the ability to write test sequences which run on every implementation of the JVM. Such a test-sequence is much more desirable as it allows multiple platforms to be tested with the one sequence.

Generation of such a test-sequence involves recording all JVM choices made by the model checker. For example, if a number of threads is blocked, then we must record that any one of the threads may be given the lock. External choices made by the application, such as thread identifier, method called, input parameter, may all be fixed.

The SAL simulator provides a programmable interface allowing such information to be extracted. Our approach is as follows.

1. Produce a counter-example for a particular test condition.
2. Extract the external choices (the first 4 columns of the table below).
3. Encode the external choices as a SAL module.
4. Extend the system with the new module to produce a new system which behaves as the test case.
5. Step through the model in the SAL simulator, extracting all traces with different JVM choices at *giveLock* and *notify* events.

For example, result of this process is presented in Table 4. The table describes a test case involving the two traces that may be observed for the test condition stated above as *th1*.

Note that threads p1 and p2 started at times 3 and 4 respectively may terminate in either order, and that the outputs finally returned will occur in the corresponding order.

	Thd	Call	In	Trace1		Trace2	
				Exit	Out	Exit	Out
1	p1	put	0	1	-	1	-
2	p2	put	2	2	-	2	-
3	p1	put	1	5	-	6	-
4	p2	put	2	6	-	5	-
5	c1	get	-	5	0	5	0
6	c1	get	-	6	2	6	2
7	c1	get	-	7	1	7	2
8	c1	get	-	8	2	8	1

Table 4. Test case for ManyProducersWaiting

5. Conclusion and future work

In this paper we developed a UML-based model to capture the underlying functionality of Java concurrency, and then extended this model to explicitly include testing aspects of the concurrent system. Test sequences were then systematically generated from this model. In the Producer-Consumer example discussed in the paper we applied the model checker SAL to generate these test sequences. This involved manually translating the UML model to the SAL model. However, this translation is systematic and we are currently investigating the use of the TefKat [4] model transformation engine to perform this transformation automatically.

The test sequences produced were executed on components of the Producer-Consumer system using the ConAn testing tool by applying a deterministic 'run to completion' execution strategy. In order to automatically create test sequences based upon the 'run-to-completion' strategy, the required constraints were combined with the SAL translation of the model.

The approach taken in this paper is an improvement on the existing specification-based testing strategies. Not only are testing aspects incorporated directly into the models, but the generation of test sequences from the models is systematic.

Acknowledgements

This research is funded by the Australian Research Council Discovery grant, DP0343877: Practical Tools and Techniques for the Testing of Concurrent Software Components, and DP0451830: Formalizing Software Design Pattern Concepts and Pattern Specifications using Metamodeling.

References

[1] Abdurazik and J. Offutt. Using UML collaboration diagrams for static checking and test generation. *Proc. of UML '2000 - The Unified Modeling Language*, LNCS 1939, pages 416-429, 2000.

[2] L. Briand and Y. Labiche. A UML-based approach to system testing, *Software and Systems Modeling*, 1(1):10-42, 2002.

[3] L. Briand, J. Cui, and Y. Labiche. Towards automated support for deriving test data from UML statecharts, *Proc. UML' 2003 - The Unified Modeling Language*, LNCS 2863, pp. 249-264, Springer Verlag, 2003.

[4] K. Duddy, A. Gerber, M.J. Lawley, K. Raymond, J. Steel. Declarative Transformation for Object-Oriented Models. In Transformation of Knowledge, Information, and Data: Theory and Applications, edited by P. van Bommel. Idea Group Publishing, 2005.

[5] R. Duke, L. Wildman, B. Long. Modeling Java Concurrency with Object-Z, *Proc. SEFM'03 – Software Engineering and Formal Methods,* IEEE Computer Society Press, pp 173-181, 2003.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2nd Edition, 2000.

[7] S-K. Kim and D. Carrington. A Formal Model of the UML Metamodel: the UML State Machine and its Integrity Constraints. Proc. of ZB 2002. LNCS. 2272, pp. 497-516, 2002.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd Edition, 1999.

[9] Long, D. Hoffman, and P. Strooper. Tool support for Testing Concurrent Java Components, *IEEE Transactions on Software Engineering*, 29(6):555-566, 2003.

[10] Long, P. Strooper, and L. Wildman. A Method for Verifying Concurrent Java Components based on an Analysis of Concurrency Failures. *Concurrency and Computation: Practice and Experience*, Wiley, submitted August 2003.

[11] L. de Moura, S. Owre, H. Ruess, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2, *Proc. CAV 2004 – Computer-Aided Verification,* LNCS 3114, 496-500, Springer Verlag, 2004.

[12] OMG, Unified Modeling Language Specification, version 1.3, 1999, http://www.omg.org

[13] OMG, UML 2.0 Specification (UML 2.0 Infrastructure Final Adopted Specification), OMG Document ptc/03-09-15, http://www.omg.org/uml/, 2003.

[14] L. Wildman, R. Duke, and P. Strooper. Viewpoint -Based Testing of Concurrent Components, *Proc. IFM'04 – Integrated Formal Methods,* LNCS 2999, 501-520, 2004.