Checking Concurrent Contracts with Aspects

Eric Kerfoot
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, UK
eric.kerfoot@comlab.ox.ac.uk

Steve McKeever
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, UK
steve.mckeever@comlab.ox.ac.uk

ABSTRACT

The applicability of aspects as a means of implementing runtime contract checking has been demonstrated in prior work, where contracts are identified as cross-cutting concerns [12, 13]. Checking contracts at runtime encounters a set of challenges within concurrent environments, such as the risk that evaluation will introduce deadlock to code which is otherwise deadlock-free. This paper presents a simple methodology for generating runtime contract checking aspects targeted at concurrent programs. The novel features of this approach allow contracts to depend on active objects without race conditions or deadlock, and addresses issues relating to timing and blame assignment. The CoJava language is discussed whose tool-supported aspect generation methodology allows the correct checking of contracts predicated on active objects.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Concurrent, distributed, and parallel languages;
D.3.3 [**Language Constructs and Features**]: Concurrent programming structures—*Classes and objects*;
F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Specification techniques

Keywords

Java, Active Objects, Concurrency, AOP, Runtime Assertion Checking

1. INTRODUCTION

Checking assertions derived from a specification at runtime is a useful formal means of testing a program's correctness. Specifications following the Design-by-Contract [21] (DbC) approach are composed primarily of type invariant predicates, method precondition predicates, and method postcondition predicates. These have been identified as cross-cutting concerns [12] which could be naturally expressed as aspect-oriented [13] advice.

In respect to Java [8] development, runtime checks can be implemented using AspectJ aspects to express specifications defined with JML [15]. However aspects meant to express contracts in sequential code may malfunction when concurrency is introduced, in particular through the Active Object design pattern [14]. Deadlock, for example, can be introduced when the evaluation of a contract requires the locking of resources that cannot be freed until the evaluation is completed. The method the contract specifies may not lock the resource, thus deadlock is created in a situation that otherwise would function correctly.

This paper outlines an implementation of the aspect-based runtime assertion checking (RAC) approach that is designed for use with active objects. It employs a number of strategies to deal with deadlock, race conditions, blame assignment, and other issues.

A typical approach employs a tool to generate the aspects from the contracts, which are then woven with a test build of the program at hand. This serves as an alternative to the *jmlc* [4] RAC tool included with the JML distribution. The methodology and tools to do this has been both patented [18] and well researched [3, 6, 19, 20, 23, 25].

The tool described in this paper implements such a methodology for the Java subset language CoJava. Its novel approach generates code necessary for the active object implementation it uses, called *threaded objects*, such that it uniquely provides concurrent features other approaches lack.

Active objects represent units of concurrency where the invocation and execution of their methods happens asynchronously. Using active objects in contracts poses new challenges when these are checked at runtime. Specifically, deadlock must not be introduced when checking contracts, nor should assertion checking in general otherwise affect the program's behaviour. The CoJava tool generates aspect code that checks contracts in a way expected for sequential code, but also augments the internal implementation of threaded objects to avert deadlock in specialized situations.

Applying RAC to active objects poses a number of issues specific to the semantics of checking contracts that this paper will examine, which are summarized here:

1. If contracts reference other active objects, does the evaluation of the contract introduce deadlock, unacceptable waiting, or other significant behaviour?
2. If a contract is violated when a method is called, how is the caller informed of this?
3. If the call results in an exception being thrown, how is the caller informed of this?

SAC'10 March 22-26, 2010, Sierre, Switzerland.

Deadlock and race conditions are prevented by construction, using the same methodology when evaluating contracts answers the first question in part. Timeout events do occur when a caller waits too long, so if and when this happens in contracts there must be a mechanism to handle this event. The second question becomes identical to the third if contract violations result in exception throws, and they are questions of how the caller is informed of such events.

CoJava's mechanism for preventing deadlock allows the querying for erroneous results, but it requires the programmer to explicitly check for such events. A less error-prone solution is needed when contracts are violated however.

This paper will answer these questions in relation to the threaded object model used in CoJava. Section 2 will outline the CoJava language, including its ownership and threaded object features. Section 3 will describe the aspect-based RAC facility, detailing how active objects are correctly used in contracts without race conditions or deadlock. Section 4 will conclude with results and future work.

2. COJAVA

CoJava represents a core subset of the Java language with many features omitted. A small language like CoJava will have a simpler formal definition and hence is more suited to formal discussions. CoJava is not so small that techniques and concepts developed for it cannot be easily extended to the full Java language.

Initially CoJava was as a means for discussing the Colleague Technique [10] that addresses invariant unsoundness in the classical DbC approach. Ownership types are also used in the language as part of this methodology for ensuring the soundness of invariants, and this was used to enforce deadlock freedom with threaded objects in [11]. The formal definition of CoJava, as well as the CoJava Tool implementing the discussed techniques, can be found at the project website: `http://devel.softeng.ox.ac.uk/cojava`.

This section gives a brief outline of the ownership type system and the Colleague Technique since they apply to how the threaded object concurrent model works. In particular the use of threaded objects changes depending on whether they are owned or not, and colleague objects introduce the possibility of deadlock when contracts are checked at runtime. These details will be significant in the next section that describes the methodology for generating RAC aspects.

CoJava is specified with a subset of the JML language. This includes some annotations for types and members, invariant and contract predicate annotations, in addition to CoJava-specific elements. Two important additional elements are applied to types to indicate special objects: The **owned** annotation that indicates owned objects, and the **threaded** annotation that indicates threaded objects.

An example class, `Counter`, is given below which demonstrates CoJava code with JML specifications:

```
class Counter {
  protected /*@ spec_public @*/
    int value,max;

  //@ invariant value>=0 && value<=max;

  //@ requires max>=0;
  //@ ensures value==0 && this.max==max;
  public Counter(int max)
    { value=0; this.max=max; }

  //@ requires value<max;
  //@ ensures value==\old(value+1);
  public void inc() { value=value+1; }

  //@ requires (value+n)>=0;
  //@ requires (value+n)<=max;
  //@ ensures value==\old(value+n);
  public void add(int n){ value=value+n;}

  //@ ensures \result==value;
  public /*@ pure @*/ int get()
    { return value; }
}
```

2.1 Ownership

CoJava's transitive ownership [1, 22] type system statically enforces encapsulation of objects. An object owns another if it creates it with the `/*@ owned @*/` annotation, or has acquired its reference from another owned type. By restricting what operations are correctly typed, an object can expose owned objects only to its owners.

The CoJava type system has the following properties that are enforced by type checking:

- Owned values cannot be assigned to non-owned variables and attributes, or vice versa.
- **this** has type `/*@ owned @*/ T` in methods of class `T`.
- Methods with owned arguments can be called, and owned attributes assigned to, only through **this**.
- Methods returning owned objects and owned attributes can only be accessed through an owned receiver.

As a result, the following properties are guaranteed by the type system for well-typed CoJava programs:

- Objects cannot be aliased by owned and non-owned references at once (excluding **this**).
- Owned objects are organized into tree structures encapsulated by their owners.
- If an object is aliased through a non-owned reference, its owned objects are inaccessible to its clients .
- Clients cannot pass owned references to an object.
- Owners can access owned objects transitively, but they cannot modify the object structures created by those objects it owns.

Thus, for example, the class `TwoCounters` can guarantee that only its owners can access its members. Consequently, its invariant can rely on owned objects safely since they will not be modified in adverse ways by arbitrary external clients. Transitive owners may still modify them, but they can be reasonably expected to adhere to the invariant's conditions since their relationship with the owned objects is much more apparent.

```
class TwoCounters {
  public /*@ owned @*/ Counter c1, c2;

  //@ invariant c1.get() == c2.get();
  ...
}
```

2.2 Colleagues

Ownership cannot be used in certain situations when an invariant of one object must rely on another that is freely

accessible to external clients. The Colleague Technique [10] is used in CoJava as a mechanism to allow invariants to do this soundly, such that any correct operation on either object (the colleagues) will not violate the other's invariant.

The technique is composed of tool-generated methods used to construct bi-directional relationships between colleagues, and a methodology for generating additional invariant conditions. The added conditions ensure that the state of an object will always satisfy the invariants dependent upon it. Colleague objects use regular attributes to alias one another rather than abstract specification variables. This makes the relationship explicit between colleagues, and allows the use of the technique in runtime checking and formal reasoning.

The Iterator design pattern [7] is an example of an object that has to closely co-operate with another, namely the data structure it iterates over. The Colleague Technique first ensures that the relationship between colleagues is constructed properly such that they alias each other through attributes bearing the **collegial** annotation. An invariant condition in one colleague that depends on the other will then be translated into a mirror invariant that is incorporated into that other type's specification.

The iterator has a simple correctness requirement stating that the data structure it iterates over must have as many items as it expects there to be. The data structure must then ensure that no correct operation will remove too many items such that some iterator's invariant is violated. This is done with the mirror invariant that states the same property from the data structure's perspective:

```
class ListIterator implements Iterator {
  private /*@ nullable
    collegial List.iters; @*/ List list;
  private /*@ spec_public @*/ int pos;
  private /*@ spec_public @*/ int last;

  //@ invariant pos <= last;
  //@ invariant list != null ==>
  //@   list.size() >= last;
  ...
}

class List {
  private /*@ collegial
    ListIterator.list; @*/ Set iters;

  // mirror invariant, generated by tool:
  // (\forall ListIterator i ;
  //    iters.contains(i); size()>=i.last)
  ...
}
```

2.3 Threaded Objects

An active object is a special instance of a class whose methods are executed in a separate thread or process from those executing its clients. Typically there exists a mechanism to characterize method calls as requests, queueing these requests, and then executing them in sequence.

The CoJava threaded object model follows this approach by using the tool's code generation facilities to produce a proxy type that encloses active objects. Objects instantiated with the **threaded** annotation are automatically enclosed in a proxy, as well as all instances of classes with the annotation. These threaded objects contain an internal queue of messages which are read by processor threads assigned to the objects when needed by the internal scheduler.

The original pattern [14] has been the subject of extensions and enhancements in subsequent research. The use of futures [9], temporary holders for method call results, has been combined to make the use of active objects more seamless with sequential code [17, 2]. Other models disallow the sharing of mutable data so avoiding data races [24, 5].

Instances of threaded types exist within their own thread context, or subsystem, which is disjoint from others. The type system ensures that mutable data is not shared by allowing only methods with certain argument and return types to be called asynchronously. A method's arguments and return value is the boundary between the threaded object and the rest of the system, hence if mutable objects cannot pass over this boundary, then race conditions do not occur.

A method may be called on a threaded receiver if its argument types and return type are *admissible*. Similarly an attribute can be accessed and assigned to if its type is admissible. An admissible type is a primitive type, threaded object type, a subtype of the `StringSerializable` interface that facilitates cloning, or an immutable type.

The annotation **immutable** can be applied to class types to designate its instances as immutable objects. The methods and attributes of such a class must abide by certain rules which guarantee the immutability of the instances.

When a method is called on a threaded receiver, the return value is an instance of `Result`, which serves as a promise object [16]. This object is used to query for the eventual result of the method call, to check if the call has completed, and to check for any errors the call might have produced.

The client can only wait for finite amounts of time for the return value to come back before the `Result` instance indicates a timeout event, hence deadlock caused by objects waiting indefinitely on each other is averted. The method `Result.objectResult()` accepts as an argument a timeout value in milliseconds, and will wait for that period of time expecting an object to be received as the result of the call that created the `Result` instance.

The Producer-Consumer example given below demonstrates these concepts. The type `StringQueue` is a threaded type that maintains a queue of `String` instances. Since these are immutable, they can be passed between threaded objects without the need to clone. The method `objectResult()` is used to query for the eventual result of the call. In the example, it will wait 100 milliseconds before return **null** and setting the internal flag to indicate a timeout has occurred.

```
/*@ threaded @*/ class Producer {
  public void produce(StringQueue q) {
    for(int c=0;true;c=c+1)
      q.add(""+c);
  }
}
/*@ threaded @*/ class Consumer {
  public void consume(StringQueue q) {
    while(true){
      Result r=q.get();
      String i=(String)r.objectResult(100);

      if(r.hasTimedOut()) // handle timeout
      else if(r.isError()) // handle error
      else  ... // consume i
    }
  }
}
...
StringQueue i=new StringQueue(10);
```

```
Producer p= new Producer();
Consumer c= new Consumer();
p.produce(i); c.consume(i);
```

Ownership is used to allow calls on owned objects without the use of `Result`. Deadlock occurs when two or more objects are related through circular aliasing relationships, in which case they may wait indefinitely on each other if a timeout mechanism is not used. Owners can wait indefinitely for owned objects to respond without the risk of deadlock due to the hierarchy imposed by ownership [11]. Owned objects can still alias their owners, directly or indirectly, through regular references but are thus obliged to use `Result`.

For example, an owned threaded instance of `Counter` can be queried directly for its current value. A **void** method still produces a `Result` object, however this can be ignored as the following demonstrates.

```
/*@ owned threaded @*/ Counter c=
    new /*@ owned threaded @*/ Counter(10);

c.inc();
int i=c.get();
```

The next section will define the aspect-generation methodology used to implement RAC testing for these features of CoJava. Special provisions have to be made to ensure that the RAC instrumentation code does not introduce adverse semantics to the program, especially in the case of threaded colleague objects.

3. ASSERTION CHECKING

This section outlines the methodology for generating aspects that implement RAC for CoJava programs. The aspects are produced by the CoJava tool rather than by hand as separate documents. Features related to the threaded objects concurrency model will also be discussed, in particular how contracts involving owned, non-owned, and colleague threaded objects are handled.

The generated aspects evaluate contract predicates and throw exceptions when they evaluate to **false** or when an exception was thrown during evaluation. CoJava does not include exceptions in its language however, so any exception throw is treated as a terminal state where the program exits with a printed stack trace.

In the general case for Java exceptions can be caught and dealt with, so this approach is feasible when applied to the full language. However, exceptions thrown due to contract violations indicate a fundamental error, in which case a program should not ever catch them and attempt to recover.

3.1 Checking An Assertion

Given some predicate P, a standardized block of code is generated by the tool to evaluate it. If an exception is thrown during evaluation, this must be wrapped in another exception that indicates the problem occurred at this stage. If P results in **false**, a different exception must be thrown that indicates where in the original code the P occurs, and states what the original text of P was. The following is the basic template for the code the tool generates:

```
boolean __c=false;
try { __c = (P); } catch(Throwable t)
{ throw new ContractEvalException
    (FILENAME,LINE,COL,t,"P");}
if(!__c)throw new PreconditionException
    (FILENAME,LINE,COL,CLASSNAME,"P");
```

The type `PreconditionException` is thrown when P is a precondition, otherwise `PostconditionException` or `InvariantException` would be used as appropriate. `ContractEvalException` indicates an error has occurred when evaluating P. These are all subtypes of `RuntimeException` which allows them to be thrown even when not mentioned in the enclosing method's **throws** clause.

3.2 Generating Advice

The contract elements checked are invariants, pre-, and post-conditions. Inter-type methods and advice are generated to check these contracts.

3.2.1 Invariants

Given a type `C`, an inter-type method called `CInvariant()` is created which contains the code for checking that type's invariant predicates. A type `D` that subtypes `C` will have a method `DInvariant()` which calls `CInvariant()` before performing its own checks.

Each class will also have an inter-type method called `checkInvariant()` which calls the appropriate invariant checking method. An interface `InvariantObject` is also defined with this method which every class is declared to implement. This allows one advice block to be defined which checks the invariant before and after every method call. If the aspect being generated is called `A`, then this advice would be produced as such, where the pointcut `colleagueHelper()` matches helper methods used by the Colleague Technique:

```
Object around(InvariantObject obj) :
    this(obj) && execution(* *(..))
    && !cflow(call(void *Invariant(..)))
    && !colleagueHelper() && !within(A)
{
  obj.checkInvariant();
  Object __result=proceed(obj);
  obj.checkInvariant();
  return __result;
}
```

The pointcut `!cflow(call(void *Invariant(..)))` is used to prevent recursion by not allowing the advice to be applied when within the flow of a method ending with *Invariant*, such as `checkInvariant()` and `CInvariant()`. The pointcut `!within(A)` prevents invariants from being checked when contract expressions are evaluated, and `execution(* *(..))` is used to match any method call.

Together these prevent recursive cases where calling methods in invariant or contract checks initiates a new invariant check, which then repeats indefinitely. It also prevents the recursive case where an invariant check performed on one object initiates a check on its colleagues, which call it back and initiate infinite recursion.

After advice is also defined which calls the `CInvariant()` method after the constructor for class `C` completes. It's important to call this method rather than `checkInvariant()` so that **super**() calls in the constructors of subtypes do not check the invariant defined in that type, which may not yet be established. After advice is also defined which calls `checkInvariant()` after any external client assigns to a public attribute.

3.2.2 Method Contracts

Method contracts are checked in around advice, rather than before advice for preconditions and after advice for postconditions as in [23, 3, 19, 20]. Having one piece of advice rather than two reduces the complexity of the generated aspects, and makes the storage of `\old()` values easier. When an expression of the form `\old(E)` is present in a postcondition, a local variable is created in the advice that is assigned `E`, which then is substituted for the original expression in the postcondition predicates.

The generated advice to check the pre- and postconditions for the method `Counter.inc()` is given in Figure 1, where the assertion checking blocks described in subsection 3.1 are omitted for brevity. Evaluation blocks described in the previous subsection are used to evaluate the pre- and postconditions and throw the appropriate exceptions when necessary. This method returns **void**, thus the advice for methods returning values would have a return value derived from `proceed()`.

3.3 Checking Contracts of Threaded Objects

This basic approach works for sequential CoJava, and by extension Java in general, with behaviour similar to the code *jmlc* [4] produces. Using threaded objects in contracts however introduces timing and deadlock issues, in particular deadlock is unavoidable when threaded colleagues check their invariants if special consideration is not taken.

Using `Result` instances in contracts is normally not possible since its result querying methods are not pure. Even if they were, `Result` is cumbersome to use in contracts since predicate methods would have to be used. Timeout events in contracts also have an ambiguous meaning, since predicates that define properties which are true given the system's current state may still wait too long for responses. The aspects generated to implement RAC employ two strategies to address these issues:

- The first solution is to allow calling active object methods directly in contracts, without the explicit use of promise objects. Code will be generated in the aspects that use `Result` implicitly with a default timeout value of one second. If a timeout event occurs, an exception is thrown to indicate this.

  This addresses the inelegant use of `Result` in contracts, and allows the implicit use of non-pure methods in a way that is ostensibly pure.

  However there still is no means to differentiate between timeouts cause by false assertions and those resulting simply from busy objects. In general it is desirable for contracts to be checked quickly, so timeout values longer than the default are not reasonable, but there also is no means to define shorter periods. It would be preferable then to not predicate contracts on non-owned threaded objects, and indeed an invariant cannot rely on these.

- Ownership allows methods to be directly called without `Result` objects being used, which never produces deadlock since the relationship between owned and owners is acyclic. Since timeout events do not occur, a thrown exception will only indicate an error in contract evaluation or a contract violation.

  However this has reduced utility since methods with owned arguments can only be called when the receiver is **this**. Ownership is therefore suited primarily to specifying invariants and internally-used methods which operate on owned threaded objects.

These two approaches allow the use of active objects in contracts such that timeout events, deadlock, and practicability are addressed. Each has its own advantages and drawbacks, so a judicious choice of which must be made depending on the situation.

3.4 Colleagues and Deadlock

Two colleague objects are related in a way such that they may soundly predicate their invariants on each other. They are also guaranteed to alias each other through regular references stored in colleague attributes. This works correctly in sequential CoJava where both colleagues are non-threaded.

If the technique is extended to threaded objects, then it is only valid when both colleagues are threaded classes, in which case it allows an invariant between units of concurrency. Methods of colleagues need to be called directly in invariants without the use of `Result` instances. Deadlock arises if one colleague calls the method of another, since the mutual aliasing between colleagues produces mutual method calls, which will wait indefinitely for results. Modifications to the message-passing and processing code is made by the generated aspects to prevent this from happening.

Consider the `StringQueue` class whose `iters` attribute aliases multiple collegial instances `StringIterator`:

```
/*@ threaded @*/ class StringQueue {
private /*@ spec_public owned @*/
    ArrayList items;
private /*@ collegial
    StringIterator.queue; @*/ Set iters;

//@ invariant maxSize>0;
//@ invariant items.size()<=maxSize;
...
public /*@ pure @*/ int size(){...}
public StringIterator stringIter(){...}
}

/*@ threaded @*/ class StringIterator {
private /*@ collegial
  StringQueue.iters; @*/ StringQueue queue;
private /*@ spec_public @*/ int pos;
private /*@ spec_public @*/ int last;

//@ invariant pos <= last && pos>=0;
//@ invariant queue != null ==>
//@          queue.size() >= last();
...
//@ ensures \result==last;
public /*@ pure @*/ int last() { ... }

//@ requires queue!=null;
//@ ensures \result == (pos<last);
public /*@ pure @*/ boolean hasNext(){...}

//@ requires hasNext();
//@ ensures pos==\old(pos)+1
//@   && \result==\old(queue.getAt(pos));
public String next(){...}
}
```

When a `StringQueue` creates a new iterator, it also must create the colleague association using helper methods gener-

```
void around(Counter obj) : this(obj) && execution(void Counter.inc(Object)) {
  boolean __check;
  // Assertion check block where P is (obj.value < obj.max)
  int oldvar0 = obj.value + 1;
  proceed(obj);
  // Assertion check block where P is (obj.value == oldvar0)
}
```

Figure 1: Checking The Contract For Counter.inc()

ated by the tool. This will ensure that the iterator references the queue through the attribute `queue` and that the iterator's reference will be stored in `iters`.

The evaluation of contracts however introduces deadlock whenever invariants are checked. For example, when `stringIter()` exits the invariant for the `StringQueue` instance will be checked, which requires calling `last()` on all iterators. Since the queue instance is busy checking its invariant, the iterators will never get a response when they call `size()` as part of the invariant check that occurs before `last()` proceeds. All method calls made when checking invariants occur without timeout mechanisms, therefore the situation where two or more objects are deadlocked waiting for one another has been reached.

The collegial relationship is a special case of mutual aliasing between objects, so aspects can be generated that augment the threaded object infrastructure to account for such situations. Two modifications to the semantics of threaded objects are introduced:

- Figure 2 outlines changes that prevent recursion. Instances of `Message` encapsulate the requests sent to threaded objects. A boolean attribute is added to this type that indicates when the message is sent by an object currently checking its contracts. Advice is also defined that match the methods of `ThreadedObjectBase`, the base type of threaded proxies, so that this flag is set when appropriate.

  If an object processes a message whose flag is set, it first calls the method `receiveInvariant()` that then calls `receiveMessage()`, which does the actual message processing. This ensures that the processing is done in the control flow of a method ending with "Invariant". Consequently further invariant checks are not performed, thus avoiding infinite recursion between colleagues who would otherwise pass invariant-checking messages back and forth endlessly.

  When the invariant of `StringQueue` is subsequently checked, the call to `last()` of every iterator will not trigger an invariant check for those objects. Such a check would produce deadlock, thus this mechanism now allows the correct checking of method specifications.

- However, if `StringQueue` calls a method of a colleague iterator within the body of a method, the iterator's invariant will be checked and will again encounter deadlock. The call to the colleague will now always indicate a timeout event in a situation that would otherwise function correctly if contracts were not checked.

  Any method call on a colleague object is a special case of a visibility state. The caller will now check its invariant, indicate that it is entering a visible state, perform the call and wait indefinitely for a response. The colleague will call a method of this object as part of its invariant check, and which point messages associated with contract evaluation must be processed concurrently. This is safe to do since the object has asserted its invariant and so is in a consistent state, and will not modify its state while it waits for a response from its colleague.

  The aspect implementing this, when a `StringQueue` instance calls a method on a colleague `StringIterator` instance, is the following where the boolean attribute `isVisible` indicates whether the current object is consistent and thus can safely be visible to others:

```
Object around(StringQueue caller,
      Threaded_StringIterator rec):
  this(caller) && target(rec) &&
  call(* *(..)) &&
  !cflow(call(void *Invariant())) &&
  if(caller.isAssociated(caller,rec))
{
  caller.checkInvariant();
  caller.__thread.isVisible=true;
  Object r=proceed(caller,receiver);
  if(r instanceof Result)
    ((Result)r).waitForResult(0);
  caller.__thread.isVisible=false;
  return r;
}
```

  Finally, an override of `sendMessage()` is defined that instructs the scheduler to perform a concurrent message process operation. This happens when the message has originated from a colleague object, but only if the current object is busy (that is processing a message) and also visible. For `StringQueue` this is given in Figure 3.

Normally allowing a threaded object to process multiple messages at once leads to race conditions, and the risk of an object being accessible when its invariant does not hold. Because the relationship between colleagues is known and controllable, it can be determined when it is safe to allow multiple messages to be processed. This is only safe when the current object has checked its invariant to ensure consistency, and then indicated that it is free for the internal thread scheduler to allow concurrent processing. This does induce a behaviour change compared to the uninstrumented code, in that the call from one colleague to the other must now block and wait for a response rather than continuing.

3.5 Reporting Errors

In a sequential setting, when a contracts evaluates to false or an error occurs, an exception is thrown and the program

```
public boolean Message.isContractCheck=false;

public void ThreadedObjectBase.receiveInvariant(Message m) { receiveMessage(m); }

void around(ThreadedObjectBase t,Message m) : this(t) && args(m)
    && execution(void sendMessage(Message)) && cflow( call(void *Invariant()) ||
            call(void ThreadedObjectBase.receiveInvariant(Message)) )
{ m.isContractCheck=true; proceed(t,m); }

void around(ThreadedObjectBase t,Message m) : this(t) && args(m) && if(m.isContractCheck)
    && !cflow(call(void *.receiveInvariant(..))) && execution(void receiveMessage(Message))
{ t.receiveInvariant(m); }
```

Figure 2: Identifying Contract-checking Messages

```
public synchronized void Threaded_StringQueue.sendMessage(Message m) {
  if(!m.isContractCheck) { super.sendMessage(m); return; } // continue normally
  StringQueue d=(StringQueue)delegate;
  if(isActive && isVisible && m.sender instanceof Threaded_StringIterator &&
      d.isAssociated(null,(Threaded_StringIterator)m.sender))
    threads.activateSingleMessage(this,m); // process concurrent message
  else
    super.sendMessage(m); // continue normally
}
```

Figure 3: Allowing Concurrent Messages

exits. In Java this exception could be caught and dealt with, although it's usually a bad idea. Either way the fact that a contract violation occurred is reported in the thread that caused it.

With threaded objects this is not the case. The sender of a message that induces a contract violation will reside in a different thread context from the receiver of the message, where the exception will be thrown. A sender cannot be expected to wait for the message to be processed to see if an exception was raised, but there must still be some way of knowing when calling code has induced an error.

By default when a message is processed by a threaded proxy that causes an exception to be thrown, an error message is sent through the `Result` instance associated with the current message, and then the exception is rethrown. This is caught by the thread assigned to that object, which prints a report to the standard error device and then attempts to continue processing messages.

The `Result` class includes the `isError()` method that returns **true** when an error has been received rather than the expected return value. When an error is received, query methods such as `objectResult()` return **null** or a default value as appropriate. The `getError()` method returns the `String` representation of the exception that was received, which can then be printed.

Unlike exceptions, using these facilities requires explicit handling on the part of the calling code, such that errors occurring during calls to other threaded objects can be ignored. Similarly timeout events have to be handled if the caller waits too long for a response. The `consume()` method of the `Consumer` class given in Section 2.3 demonstrates this in practise.

Checking for timeout and error events is necessary so that blame for a precondition violation, for example, will fall on the caller and not be ignored. Calls to owned objects that return values must block and do not produce a `Result` object, so when an error is received in these situations, it is immediately rethrown.

Explicit checks or exception throws from owned calls are important to replicate the behaviour found with sequential code, where consequences of an incorrect call are immediately evident at the point of call rather than later in the program's execution. This is important in ensuring that the caller is "blamed" for the incorrect call, thus pinpointing the source of the error as that operation, rather than a future one that discovers the system to be inconsistent.

Threaded objects introduce the complication that a call and its consequences occur in different threads and at different times during execution. Callers must take care to explicitly assume blame for erroneous calls so that the source of the problem can be identified. The aspect checking code is generated with this in mind, where the throwing of exceptions to indicate contract violations still allows threaded objects to attempt recovery while indicating these events through the `Result` instances.

4. CONCLUSION

This paper has presented the aspect-based assertion checking mechanism used with the CoJava language. Using the tool-generated aspects allows the checking of JML contracts at runtime whose behaviour and construction are relatively straight-forward with sequential code. When concurrency is introduced with threaded objects, the evaluation of contracts at runtime can introduce deadlock if special cases of object relationships are not taken into account.

The same architectural choices that ensure deadlock freedom by construction, that is the required use of promise objects and ownership, ensure that contracts do not normally introduce deadlock or otherwise significantly impact the behaviour of running programs. The special case of colleague objects is handled by specialized aspects that modify the underlying implementation of threaded objects.

The current CoJava implementation is not optimized for efficiency or fairness between threads. Future implementations of active objects that use the techniques described here may employ more efficient mechanisms and so have very different architectures. The RAC approach described here can be adapted by migrating the intent of the aspects the tool generates, in particular the solution to colleague deadlock is to allow concurrent messages only in certain safe situations.

Active objects represent a simple and coherent methodology of introducing concurrency to modern object-oriented languages. They can abstract away concepts of threads, data locks, and synchronization, making it easier for design and implement highly concurrent applications. This paper has discussed as powerful technique for testing such designs at runtime in a rigorous manner, extending the ideas JML has pioneered successfully with sequential Java programs.

5. REFERENCES

[1] P. S. Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32, 1997.

[2] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[3] S. Balzer, P. T. Eugster, and B. Meyer. Can aspects implement contracts? In *In: Proceedings of RISE 2006 (Rapid Implementation of Engineering Techniques)*, pages 13–15, 2006.

[4] Y. Cheon and G. Leavens. A runtime assertion checker for the java modeling language. In *International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada*, pages 322–328. CSREA Press, June 2002.

[5] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal ownership for active objects. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 139–154, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Y. A. Feldman, O. Barzilay, and S. Tyszberowicz. Jose: Aspects for design by contract. *Software Engineering and Formal Methods, International Conference on*, 0:80–89, 2006.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] J. Gosling et al. *The Java Language Specification*. GOTOP Information Inc., 5F, No.7, Lane 50, Sec.3 Nan Kang Road Taipei, Taiwan, 1996.

[9] R. H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[10] E. Kerfoot and S. McKeever. Maintaining invariants through object coupling mechanisms. In T. Wrigstad, editor, *3rd International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO), in conjunction with ECOOP 2007*, Berlin, Germany, July 2007.

[11] E. Kerfoot, S. McKeever, and F. Torshizi. Deadlock freedom through object ownership. In *IWACO '09: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, pages 1–8, New York, NY, USA, 2009. ACM.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[14] R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc.Pattern Languages of Programs,*, 1995.

[15] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[16] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM.

[17] K.-P. Löhr and M. Haustein. The JAC system: Minimizing the differences between concurrent and sequential java code. *Journal of Object Technology*, 5(7), 2006.

[18] C. Lopes, M. Lippert, and E. Hilsdale. Design by contract with aspect-oriented programming, 2002. U.S. Patent No. 06,442,750, Issued August 27,2002.

[19] C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *In Proceedings of the 22nd international conference on Software engineering*, pages 418–427. ACM Press, 2000.

[20] D. H. Lorenz and T. Skotiniotis. Contracts and aspects. Technical Report NU-CCIS-03-13, College of Computer and Information Science, Northeastern University, Boston, MA 02115, Dec. 2003.

[21] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[22] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[23] H. Rebêlo, S. Soares, R. Lima, L. Ferreira, and M. Cornélio. Implementing java modeling language contracts with aspectj. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 228–233, New York, NY, USA, 2008. ACM.

[24] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *European Conference on Object Oriented Programming ECOOP 2008*, 2008.

[25] D. Wampler. Contract4J for design by contract in Java: Design pattern-like protocols and aspect interfaces. In Y. Coady, D. H. Lorenz, O. Spinczyk, and E. Wohlstadter, editors, *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 27–30, Bonn, Germany, Mar. 20 2006. Published as University of Virginia Computer Science Technical Report CS–2006–01.