Concurrent Contracts for Java in JML

Wladimir Araujo
Juniper Networks
700 Silver Seven Road
Ottawa, ON, K2V 1C3, Canada
waraujo@juniper.net

Lionel Briand
Simula Research Laboratory
and University of Oslo
P.O. Box 134, Lysaker
1325 Norway
briand@simula.no

Yvan Labiche
Software Quality Eng. Lab.
Dept. of Systems and Computer Eng.
Carleton University
1125 Colonel By Drive
Ottawa, ON, K1S 5B6, Canada
labiche@sce.carleton.ca

Abstract

Design by Contract (DbC) is a software development methodology that makes use of assertions to produce better quality object-oriented software. The idea behind DbC is that a method defines a contract stating the requirements a client needs to fulfill to use it, the precondition, and the properties it ensures after its execution, the postcondition.

Though there exists ample support for DbC for sequential programs, applying DbC to concurrent programs presents several challenges. The first challenge is interference, the product of multiple threads of execution modifying and accessing shared data. The second is the specification of thread-safety properties in the presence of inheritance.

We present a solution to these challenges in the context of Java programs by extending the Java Modeling Language (JML) specification language. We experiment our solution on a large size industrial software system.

1 Introduction

Including specifications of program behaviour together with the source code is not a new idea. Design-by-Contract (DbC) [17] is one of the most elaborate software development methodologies that put such idea in practice. Following DbC principles, a method defines a contract stating the requirements a client needs to fulfill to use it, the precondition, and the properties it ensures after its execution, the postcondition. Contracts can be treated as assertions about the state of a program at a certain point. A program can be instrumented with code that checks the validity of the assertions at runtime and upon failure throws an exception indicating where it happened. DbC also defines object invariants, properties that must hold in all visible states of an object. The visible states of an object are the states just after object construction, just before a visible method execution, and just after a visible method execution. Behavioural subtyping [2, 7, 16] is an integral part of DbC. A subtype automatically inherits the specification (contracts and invariants) from its super-types [9]. The effective precondition of a method is the disjunction of all the inherited preconditions and the method's declared preconditions. The effective postcondition is the conjunction of all inherited postconditions for which the associated precondition is satisfied and the method's declared postconditions if associated preconditions are satisfied. The effective class invariant is the conjunction of all inherited class invariants with the object's declared invariant. This guarantees that a subtype can be properly used in place of its super-type(s).

The Java Programming Language [4] does not provide native support for DbC. The Java Modeling Language (JML) [13, 15] is a specification language used to write contracts. It includes notations for pre- and postconditions, invariants, and offers mechanisms for specification inheritance, thus providing support for the DbC paradigm. The JML toolset comes with a compiler that translates specifications into runtime assertion checking (RAC) code, producing Java classes instrumented with executable assertions. The JML compiler [7] produces RAC code that enforces behavioural subtyping.

Most work on DbC focused on sequential programs, and applying DbC to concurrent programs presents several challenges. The first challenge is interference, the product of multiple threads of execution modifying and accessing shared state. As further discussed below, this may cause RAC code to incorrectly report errors during correct execution and vice-versa. Solving the issue of interference with a focus on runtime assertion checking is one major contribution of this work. The second challenge is the specification of thread-safety properties in the presence of inheritance. We argue that, contrary to what is currently done in the literature, thread safety properties cannot be specified in preconditions in the presence of inheritance.

The rest of the paper is structured as follows. Section 2 describes these challenges in detail. Section 3 presents our solutions. Related work is discussed in section 4. We then present an industrial case study (section 5). We conclude in section 6.

2 Contracts and Concurrency

This section presents the problems of using contracts to specify behaviour and generate runtime assertion checking code for concurrent programs. Although we use Java and JML, the principles we introduce would likely, for the most part, apply to other programming languages. We do not describe JML in detail but briefly describe the constructs we use to illustrate the challenges of concurrent contract specification and verification.

2.1 The Problem of Interference

Two threads interfere when one unintentionally changes data the other observes. This becomes a problem if, due to an arbitrary interleaving, one thread's perception of the shared data is not true due to a modification made by another thread and it relies on such perception for future computations.

Before continuing the discussion, we must recall the notions of a method's pre-state and post-state [13]. "The *pre-state* of a method call is the state just after the method is called and parameters have been evaluated and passed, but before execution of the method's body. The *post-state* of a method call is the state just before the method returns or throws an exception; in JML we imagine that `\result` and information about exception results is recorded in the post-state" ([13], p. 8).

The method specification in Figure 1 (excerpt from [19]), composed of two specification cases separated by the keyword **`also`** (each with a precondition and the corresponding *expected postcondition*, the postcondition to be established if the precondition is satisfied), simply tells that the head of the list will move to the next element and the method will return the value of what used to be the first element of the list if the list is not empty (lines 5-9), and returns **`null`** otherwise (lines 1-4). In JML, the preconditions of a method (i.e., the **`requires`** clauses), as well as arguments to the `\old` operator in postconditions are evaluated in the method's pre-state. The method postconditions (i.e., the **`ensures`** clauses) are evaluated in the method's post-state.

Although straightforward, this specification is not correct in a multi-threaded environment. Suppose that `extract()` is invoked by thread 1 and in the method's pre-state, `head` references the same object as `last` (i.e., the list is empty). Suppose, also, that thread 2 pre-empts thread 1 right after thread 1 acquires the lock on **`this`** to fully execute method `insert()`, which does not acquire such lock for performance reasons. The postcondition of `insert()` specifies that `head` is not referencing the same object as `last`, i.e., the list is not empty. Once thread 1 resumes execution and acquires the lock on `head`, it will return the first element of the list, violating the postcondition of `extract()` for an (expected) empty list, i.e., that it should have returned **`null`**.

This is an example of interference in the context of DbC. This problem is not specific to Java or JML. Any object-oriented language in which the scenario we described above is realizable and provides support for DbC via runtime assertion checking (RAC) is prone to this problem. It is important to emphasize that such problem arises due to the combination of DbC and the program under execution. It is not due to erroneous concurrency control on the part of the implementation either of the client or the provider. Interference can also happen between the contract evaluation point (pre- and post-state) and the method entry and exit points. Since interleaving occurs outside the method body, this is called *external interference*. The previous case, where interleaving occurs inside the method body is called *internal interference*.

```
public class LinkedQueue {
    protected /*@ spec_public @*/ LinkedNode head;
    protected /*@ spec_public @*/ LinkedNode last;
    //@ public invariant head.value == null;
1   /*@ public normal_behavior
2     @  requires head == last;
3     @  assignable \nothing;
4     @  ensures \result == null;
5     @ also public normal_behavior
6     @  requires head != last;
7     @  assignable head, head.next.value;
8     @  ensures head == \old(head.next) &&
9     @   \result == \old(head.next.value);
10    @*/
    public synchronized Object extract() {
        synchronized (head) {
            Object x = null;
            LinkedNode first = head.next;
            if (first != null) {
                x = first.value;
                first.value = null;
                head = first;
            }
            return x;
        }
    }
}
```

Figure 1. Method `extract()` of class `LinkedQueue`

2.2 Specification Inheritance

Specifications can be inherited from interfaces and super-classes, and the issues of data abstraction and concurrency in the context of inheritance are the same in both cases. In the following we use an interface example to illustrate those issues, without loss of generality.

Figure 2 shows the specification of interface `Channel`. It declares two model fields (lines 2-3): A model field (a field with the **`model`** modifier) is a field that does not have to be implemented but can be used in a specification as any other field. Model fields allow abstract modeling [8] and play a vital role in information hiding, modular reasoning and behavioural subtyping [14]. Both fields in this example are marked **`instance`**: they are fields of the

object implementing the interface instead of static fields of the interface (JML allows instance fields on interfaces).

Interface `Channel` can be implemented by class `PipedChannel` (Figure 3) with the help of a `Pipe` (not shown). The **`represents`** clause maps the value of a model field to an expression based on concrete fields of the class: e.g., the value of model field `connected` comes from concrete fields `closed` and `remoteClosed` according to the Boolean expression in line 3 (Figure 3).

```
   public interface Channel {
1  /*@
2    public instance model boolean connected;
3    public instance model int nPending;
4    public invariant nPending >= 0;
5    public constraint connected==>\old(connected);
6    @*/
7  /*@
8    public normal_behaviour
9     requires nPending>0 || (nPending==0 &&
connected);
10    ensures
(\result!=null==>nPending==\old(nPending)-1)
11      && (\result == null ==> \old(nPending)==0);
12   also
13   public exceptional_behaviour
14    requires !connected && nPending == 0;
15    signals_only NotConnectedException;
16   @*/
   public Message receive()throws …;
   }
```

Figure 2. Interface `Channel` (excerpt).

Since method `receive()` in class `PipedChannel` does not declare any specification, it inherits the parent one without any change (i.e., the one in Figure 2), which specifies that `receive()` will return any `Message` objects the channel contains even if it has already been closed (line 10) or **`null`** if it is empty (line 11). The exceptional specification (lines 14-15) states that `receive()` will throw a `NotConnectedException` if the channel has been closed and it is empty.

The problem in this case is very similar to the previous ones: interference causes pre- and postconditions to be evaluated at unsafe points since they reference the object's internal state. The difference is that such internal state is made visible through model fields in the interface specification. Although one might argue that such specification is improper for a concurrent environment because it was not designed with concurrency in mind, nothing in the interface states that it actually is supposed to be used only in a sequential environment. One might implement it sequentially or concurrently offering the same guarantees.

This issue is more subtle than in the previous cases. Model fields are used to specify behaviour without giving out implementation details. Model fields do not have storage, i.e., their values are derived from the object's state (e.g., `nPending`'s value comes from the evaluation of `getSize()` on field `in`).

A solution to the issue of contracts for concurrent objects must accommodate the usage of model fields. Unfortunately, as far as we know, the current literature does not even acknowledge the existence of a problem. Since clients do not have visibility on how the provider realizes model fields, they cannot know which locks to acquire, and might not even know that there are concurrency control issues. Therefore, the client cannot be required to perform additional concurrency control simply to guarantee thread-safe access to fields present in specifications, as suggested in [11, 12]. This, instead, must be the responsibility of the provider.

```
  public class PipedChannel implements Channel {
    protected final Pipe in, out;
    private volatie boolean closed = false;
    private volatie boolean remoteClosed = false;
1  /*@
2    @ private represents connected <-
3    @                    !closed && !remoteClosed;
4    @ private represents nPending <- in.getSize();
5    @*/
   public Message receive()throws … {
     if(closed)
        throw new NotConnectedException();
     if(in.isEmpty()) {
       synchronized(this) {
         if(remoteClosed) {
           closed = true;
           throw new NotConnectedException();
         }
       }
       return null;
     }
     return in.take();
   }
  }
```

Figure 3. Class `PipedChannel` (excerpt).

2.3 Thread Safety

The core idea behind thread safety is one of non-interference [19]. Thread safety can be achieved in a variety of ways, which all relate to the way data can be accessed. Data that is local to a thread (i.e. not visible to other threads) is not subject to any interference issue. Access to shared data (i.e. visible to multiple threads) must be protected by a lock. By doing so, one guarantees the absence of interference when accessing such data.

JML provides several constructs to specify these aspects of thread safety [19]: e.g., the `\thread_safe` predicate specifies that a given object is thread-safe, i.e. it is either local to a thread or access to it is protected by a lock. JML provides a number of constructs to explicitly deal with locks. The `\lockset` expression refers to the set of locks held by the current thread. Although described in [19], these constructs were not implemented in the JML compiler, thus not producing any RAC code. Note that stating thread safety using `\thread_safe` is equivalent to stating it explicitly through locking requirements. The choice is based on ease of specification only.

The contract for `Pipe.get()` can be extended to include this clause as shown in Figure 4. In JML, locking and thread-safety properties are currently specified in the

precondition of a method (e.g., lines 1 and 5 in Figure 4). This presents two major problems. The first problem is the evaluation point. DbC specifies that preconditions be evaluated prior to the first statement of the method body. In JML, this is done in the pre-state. Alternatively, a design decision for JML could have been to check the pre-condition as the first statement of the method. There is no difference between these two approaches for sequential programs, i.e., when verifying functional properties. For concurrent programs, however, there is a difference. The first statement of a method might already be protected by a monitor lock (**`synchronized`** methods in Java), as illustrated by method `get()` of class `Pipe` (Figure 4). Its precondition specifies locking and functional properties. If one decides to evaluate the precondition of `get()` in the method pre-state it is subject to external interference because the thread does not yet hold a lock on **`this`**, which protects the access to the field `closed` from race conditions. If, however, the evaluation happens just before the first statement of the method, i.e., right after acquiring the lock on **`this`** but before executing any statement of the method body, interference is not a problem. However the term `!\lockset.has(this)` (i.e., the thread must not already hold a lock on the object) will evaluate to **`false`**, which is not the desired behaviour. We argue that this suggests that locking and thread-safety properties do not belong in the precondition.

The second problem is related to specification inheritance and behavioural subtyping. Since the actual precondition of a method is the disjunction of the inherited preconditions and the ones the method defines, the actual precondition can be true even if the locking or thread-safety requirements the method defines do not hold, as long as the inherited preconditions (which may not contain locking or thread-safety requirements) hold. As a result, although the method requires some locking mechanisms to exist (as defined in its precondition), no locking may in fact exist but this will not be detected as a precondition violation at run time. Conversely, a subtype may weaken the inherited preconditions in a way that inherited locking and thread safety properties are not required to hold. We argue that this is the main point for not having locking and thread safety properties in the precondition. An analogous argumentation can be made for postconditions.

The root cause of these problems is the combination of thread safety behaviour specification with functional specification. Functional specifications specify properties that must hold on the states preceding and following a method execution, i.e., on state transformations. Thread safety specifications deal with the properties that must hold to ensure that such state transformations occur as specified in a concurrent environment. A program can, then, be seen as the combination of two aspects: the functional and the concurrent aspects. The functional aspect is the one that deals with retrieving, processing and outputting data, whereas the concurrent aspect is the one that deals with the mechanisms to guarantee that such manipulations of data by multiple threads is controlled. These aspects are orthogonal, and one usually states thread safety requirements independently from functional properties. We believe that method specifications should reflect this independence.

```
 /*@ public normal_behaviour
1     requires closed && !\lockset.has(this);
2     ensures \result == null;
3    also
4    public normal_behaviour
5     requires !closed && !\lockset.has(this);
6     ensures \result != null;
 */
public synchronized Message get() throws … {
    // Body omitted
}
```

Figure 4. Method `get()` of class `Pipe`.

3 Specifying Contracts in the Presence of Concurrency

In this section we present our solution to the issues discussed previously. The solution is decomposed into two aspects: first we introduce the concept of safepoint, that is, code locations where precondition or postcondition predicates can be safely evaluated (section 3.1). Then we show how concurrency related predicates can be specified orthogonally to functional predicates (section 3.2). Section 3.3 then shows that these two solutions address the interference issue. All the new constructs introduced in this section were incorporated into the JML toolset including the JML compiler, which is the tool that generates RAC code from JML annotations. However, we do not discuss the techniques for generating RAC code for the new constructs due to size constraints.

3.1 Safepoints

A *safepoint* is any point inside the method body where it is safe to evaluate precondition, postcondition and invariant predicates. A *precondition safepoint* is a point where it is safe to evaluate preconditions and invariants, and the pre-state predicates of postconditions. A *postcondition safepoint* is a point where it is safe to evaluate the expected postconditions and the invariants. Notice that no guarantees are made with respect to postcondition formulas, present in the method specification, that are not safely evaluated at a given postcondition safepoint. Any method execution path (from the pre-state to the post-state) can have only one precondition safepoint and only one postcondition safepoint. If no precondition (resp. postcondition) safepoint is explicitly specified for an execution path, it defaults to the method pre-state (resp. post-state). In a

precondition safepoint, all preconditions, invariants and pre-state predicates are required to be safely evaluated. In a postcondition safepoint, the postconditions and all invariants are required to be safely evaluated. We propose the addition of the **`requires_safepoint`** and **`ensures_safepoint`** labels to JML to demarcate those safepoints.

3.1.1 Safepoints and Interference.

Figure 5 shows an example of their use (access to `last` is also protected by a lock on `head`). At the precondition safepoint (line 13), all the objects referenced by both **`requires`** clauses (lines 2 and 6) and the contents of the `\old` statements in the **`ensures`** clauses (lines 8-9) are properly protected by locks. At the postcondition safepoint (line 21), the field `head`, present in the **`ensures`** clause at lines 8-9, is properly protected by a lock. Since `\result` refers to local variable `x`, which in turn points to an object no longer referenced by the list, it is also thread-safe at the postcondition safepoint[1]. Finally, the object invariant can be safely evaluated both in the pre- and postcondition safepoints since it refers to `head`, which is properly locked in both places.

```
public class LinkedQueue {
    protected /*@ spec_public @*/ LinkedNode head;
    protected /*@ spec_public @*/ LinkedNode last;
    //@ public invariant head.value == null;
1   /*@ public normal_behavior
2     @  requires head == last;
3     @  assignable \nothing;
4     @  ensures \result == null;
5     @ also public normal_behavior
6     @  requires head != last;
7     @  assignable head, head.next.value;
8     @  ensures head == \old(head.next) &&
9     @    \result == \old(head.next.value);
10    @*/
11    public synchronized  Object extract() {
12        synchronized (head) {
13            //@requires_safepoint:
14            Object x = null;
15            LinkedNode first = head.next;
16            if (first != null) {
17                x = first.value;
18                first.value = null;
19                head = first;
20            }
21            //@ensures_safepoint:
22            return x;
23        }
24    }
25}
```

Figure 5. Method `extract()` of class `LinkedQueue` using safepoints to avoid internal interference.

Having the precondition of a method evaluated inside the method is counter-intuitive. A precondition, as initially presented by Meyer, can be evaluated just before entering the method or just after (i.e., before any statement of the method body is executed). These two views are equivalent because nothing *significant* to the evaluation of the precondition predicates happens between these two stages. The same idea applies to the precondition safepoint: nothing significant to the evaluation of the precondition predicate should happen between the method's pre-state and the safepoint. Then, the precondition can be evaluated either in the pre-state or at the precondition safepoint yielding the same result. The only difference is that at the precondition safepoint the method is interference-free (due to the acquisition of some locks, in this example).

Analogously, as long as nothing significant to the evaluation of the postcondition happens between the postcondition safepoint and the post-state, evaluating the postcondition in those two places is equivalent but the former is safer (no interference).

3.1.2 Safepoints and Specification Inheritance.

Recall that assertions checking specification inheritance are subject to interference because of the evaluation of model fields from concrete fields. To ensure safe evaluation of inheritance contracts, the evaluation of model fields needs to occur (i) at a location where the concrete fields are protected from interference or (ii) after their values have already been obtained and are guaranteed not to change.

Figure 6 illustrates how safepoints can be used to prevent external interference for method `receive()` of class `PipedChannel` (Figure 3), which inherits its specification from its interface (Figure 2).

An example of the first location is the safepoint at line 8 (Figure 6): Access to `remoteClosed` is protected by a lock on **`this`**, which guarantees that the value observed by the precondition is consistent with the one observed by the method. An example of the second location is the safepoint at line 3: It is located after `closed` evaluates to

```
    /* same specification as in Figure 3 */
1   public Message receive() throws …{
2     if(closed) {
3       //@ requires_safepoint:
4       throw new NotConnectedException();
5     }
6     if(in.isEmpty()) {
7       synchronized(this) {
8         //@ requires_safepoint:
9         if(remoteClosed) {
10          closed = true;
11          throw new NotConnectedException();
12        }
13      }
14      return null;
15    }
16    //@ requires_safepoint:
17    //@ ensures_safepoint:
18    return in.take();
19  }
```

Figure 6. Method `receive()` of class `PipedChannel` equipped with safepoints.

[1] The postcondition safepoint must be the **`return`** or **`throw`** statement. Additionally, the return (or throw) expression must be side-effect free, which can be easily checked at compilation time. In case the method does not return a value, the `ensures_safepoint` can be placed at the end of a block or just before the method returns.

true, which never changes afterwards.

Locking is not required in this case because Java guarantees that accesses and assignments to variables of type **int** and **boolean** are atomic. Furthermore, the **volatile** modifier guarantees that the observed value is the current value of the variable instead of a cached image. Otherwise, some locking would be required to guarantee atomic access to these fields.

3.1.3 Discussion.

Safepoints demarcate points in the code where the data the pre- or postcondition predicates observe are the same as the ones the method execution observes. The idea is not to add extra concurrency control just for the sake of evaluating contracts since it would cause the instrumented code to execute differently, likely preventing harmful interleavings present in the original code from occurring, which could lead to undetected faults on the final (uninstrumented) program. Instead, the idea is to find the right place in the code to evaluate the contract following, to the maximum extent possible, the intent of the designer, who was certainly not thinking that such predicates were all safely observable immediately before or after the method execution. For instance, when one specifies the behaviour of `LinkedQueue.get()` (Figure 1) to "return null if the list is empty or the element at the head, otherwise" one is implicitly thinking "once the list can be safely manipulated, return null if it is empty or the element at the head, otherwise".

In this light, we consider safepoints not as part of the contract but as being part of the implementation. For example, notice that the contracts of `receive()` have not changed from Figure 3 to Figure 6. Thus, correctly placing the safepoint is an implementation problem not a contract design problem. This approach is in line with the idea behind DbC, namely that a contract specifies the observable behaviour of a method without getting into the details of its implementation.

Of course, the placement of safepoints needs to follow the rules we defined earlier to guarantee the expected semantics of a contract, i.e. that it would evaluate the same in a sequential environment with or without safepoints: preconditions and postconditions are evaluated only once for a method execution and that nothing significant (to the evaluation of the predicates) happens between the beginning (end) of a method for precondition (postcondition) safepoints. [2]

Significant statements are those that can be observed through changes on the program state. We, then, define that only *unobservable statements* are allowed between the beginning (end) of the method and the precondition (postcondition) safepoint. Unobservable statements are assignment to local variables, pure method or constructor calls, loops and branching statements, try blocks, JML annotations, assignments to local ghost variables, and acquiring and releasing a lock[3].

3.2 Specification of Thread-Safe Behaviour

As we argued in section 2.3, thread safety behaviour specification and functional specification are different. Separating the two requires the introduction of new constructs to JML (section 3.2.1) and revisiting inheritance specification of concurrency aspects (section 3.2.2). An example is introduced in section 3.2.3.

3.2.1 New constructs for Thread-Safety Specification.

First, we add the **concurrent_behaviour** clause to a method specification to separate functional (e.g., JML **normal_behavior** and **exceptional_behavior** clauses) from thread-safety property specifications. A **concurrent_behaviour** specification can contain one or more of the following clauses:

- **requires_locked**, **requires_unlocked**, **ensures_locked**, **ensures_unlocked**: These specify the set of lock objects that are held or not by the current thread in the method pre-state and post-state, respectively. Null references are ignored. These replace the JML `\lockset` (part of the functional aspect in JML).
- **requires_thread_safe**, **ensures_thread_safe**: These specify that all the objects provided as argument satisfy the `\thread_safe` predicate in the method pre-state and post-state, respectively. Null references are ignored.

These clauses take a *spec-expression-list* (see section A.8 of [15]) as an argument (i.e. a comma-separated list of JML expressions). Each such expression evaluates to an object reference. These clauses all default to `\not_specified`. The **requires_...** clauses must be satisfied in the method's pre-state. The **ensures_...** clauses must be satisfied in the method's post-state.

3.2.2 Inheritance Specification.

We showed in section 2.3 that specification inheritance rules for concurrency aspects cannot be the same as for functional aspects. The semantics of specification inheritance on the concurrent aspect must be identical to the one of invariants (conjunction). The effective specification (in a subclass) of any of the new clauses is the set of reference objects resulting from the union of the argument set specified on the target object with the argument sets of its immediate supertypes. (`\not_specified` is treated as the empty set.) In other

[2] These constraints can be checked at compile time (e.g., by adapting rules of definite assignment [4]).

[3] Acquiring a lock is not observable because, in a deadlock-free program, all lock acquisitions eventually succeed [3].

words, thread-safety specifications, like invariants, should only be strengthened by sub-types.

At first, one might argue that such properties should not be inherited at all since concurrency control is very particular to a type. One would not see these properties as exposed functionality but as implementation details. However, we believe it should be possible for a class (or interface) to specify these properties for extension purposes. A typical example is a class in a messaging framework. The framework could define a `Message` interface and state locking and thread-safety properties for implementers so that they can be handled without risk of deadlocks or race conditions. Additionally, there are cases where interfaces are purposefully underspecified with respect to these properties to allow concrete implementations the freedom to choose their concurrency control strategy. This makes sense when there are two or more interfaces to be implemented and these must work in tandem. For instance, a particular message processor (implementing interface `Processor`) processes RPC messages (implementing interface `Message`). This processor can require stronger properties than the ones from `Message` since it knows it will only process RPC messages.

The semantics of the new clauses allow the specification of thread-safety and locking properties in the presence of specification inheritance. It decouples concurrency related properties from functional properties giving concurrent contracts their intuitive (expected) meaning. We do not make any claim with respect to the modularity of the concurrent aspect. This is outside the scope of this work. Our additions, however, do not disturb modular reasoning on the functional aspect of a method specification since we do not change the way specification inheritance is implemented for this aspect.

3.2.3 An Example.

As an example, consider the declaration of method `sendAndWait()` of class `PipedChannel` in Figure 7. It shows examples of some of the new clauses in lines 8-10. The **`requires_thread_safe`** clause specifies that object `r` must be thread-safe in the method pre-state. Similarly, **`ensures_thread_safe`** specifies that the object returned by the method must be thread-safe on the method's post-state. (See below for additional comments on this example.)

3.3 Thread Safety + Safepoints = No Interference

Due to space limitations this section presents an informal justification that both the thread safety mechanisms we introduced and safepoints are necessary to avoid interference. For a complete justification see [3]. We analyze a single example that is representative of the

```
/*@
1   normal_behaviour
2    requires connected && r.isRequest();
3    ensures \result.isResponse();
4   also
5   exceptional_behaviour
6     requires !connected && r.isRequest();
7     signals_only NotConnectedException;
8   concurrent_behaviour
9     requires_thread_safe r;
10    ensures_thread_safe \result;
 */
public Message sendAndWait(Message r) throws … {
11   synchronized(in) {
12       synchronized(this) {
13           //@ requires_safepoint:
14           if(closed || remoteClosed)
15               throw new NotConnectedException();
16       }
17       out.put(r);
18       return in.get();
19   }
}
```

Figure 7. Method declaration exemplifying the use of thread-safety specification clauses.

case in which predicates refer to method parameters and internal state.

Figure 7 presents a simple contract in which the effective precondition involves predicates on method parameters. The effective precondition, accounting for normal and exceptional behavior of the method is `r.isRequest()` (the disjunction of preconditions from both specification cases simplifies the terms `connected` and `!connected`): it is not simply **`true`**. In this situation, safepoints alone cannot guarantee the thread-safe observation of this predicate since `r` is external to the provider. Since it is the responsibility of the client to establish such predicate, it is also reasonable to expect that the client ensures the provider can safely evaluate it; otherwise it would be pointless to establish a state knowing it could asynchronously change before it could be observed. This is reflected by the use of the **`requires_thread_safe`** clause on all objects involved in the effective precondition (line 9): `r` is required to be thread-safe. Once such objects are thread-safe, predicates involving them can be checked at precondition safepoints since they will not change between the method pre-state and the safepoints.

A similar discussion can be made for postconditions. If such predicates involve any state, parameter or return value that must be established by the provider and observed by the client, the associated objects must be flagged as thread-safe (Figure 7, line 10). This is required because the normal postcondition (i.e., identified by keyword `normal_behaviour` on line 3) refers to the return value. For the client to observe this predicate (`\result.isResponse()`), the provider needs to guarantee that the referred object is interference free.

The precondition safepoint on Figure 7 demarcates the interference-free evaluation point for the precondition involving model field `connected`. Being protected by

locks on `in` and **`this`**, the safepoint can safely observe fields `closed` and `remoteClosed` as well as guarantee that `in` will not be asynchronously closed until the execution of `in.get()`. The safepoint guarantees absence of interference with respect to model field `connected`. Thread-safety predicates alone cannot guarantee the safe evaluation of these fields since this is achieved only during a specific point inside the method body and these predicates are applicable throughout the method execution (see section 3.1.1). Postcondition safepoints are not necessary since postconditions do not refer to any internal state.

The **`*_thread_safe`** clauses guarantee freedom from interference with respect to `r` from the method pre-state up to the precondition safepoint and with respect to `\result` on the post-state. Precondition safepoints prevent interference related to model field `connected`. As these are the only possible sources of interference, we conclude that combining safepoints and thread-safety predicates guarantees `sendAndWait()` and its contract are interference-free.

The conclusions from this example can be generalized since its contract possesses all features of JML relevant to thread-safety. It uses safepoints, thus covering references to internal state and interference, both internal and external. It also contains multiple specification cases, thus covering specification inheritance (specification inheritance is a particular case in which specification cases come from supertypes instead of being explicitly placed on the target type). It specifies predicates on method parameters and return value, thus covering predicates on external state. We can then conclude that, in general, the combination of thread-safety requirements on data to be observed by the provider and the client with safepoints (for safe evaluation of predicates referring to internal state) is required to guarantee freedom from interference.

4 Related Work

Flanagan and Freund [10] describe Atomizer, a dynamic checker for Java programs. Atomizer checks for atomicity, a fundamental property of concurrent programs. "A method is atomic if its execution is not affected by and does not interfere with the concurrently executing threads." They report on the use of atomizer on moderate size (up to 90000 lines of code) programs. Agrawal et al [1] describe a combination of runtime and static analysis to check for atomicity.

Nienaltowsky and Meyer [18] present an interesting proposition regarding the use of contracts in a concurrent environment. They target SCOOP [5], an extension of the Eiffel language to provide support for concurrency. They transform preconditions referring to separate objects (objects not owned by the current thread) in wait conditions that must be eventually satisfied once the current thread acquires a lock on such objects. Postconditions are treated similarly: locks on separate objects are not released until postconditions are satisfied. The SCOOP model does not allow invariants to refer to separate objects, so their evaluation does not cause waiting. This is equivalent to specifying all such objects as `\thread_safe` and requiring that all locks be acquired prior to calling a method. Their proposal, however, does not contemplate the full intricacies of the Java concurrency model with its synchronized blocks, multiple lock acquisitions and releases inside a method, and no restrictions on method calls on objects accessible by multiple threads.

Jacobs et al [11, 12] present a very interesting approach to the problem of concurrency control of aggregate objects. However, their solution implies that preconditions and postconditions can only refer to thread-safe data. Our approach solves this issue with the introduction of safepoints, to allow the specification of the internal behaviour of objects.

To the best of our knowledge, this work is the first to allow the specification and dynamic verification (i.e. runtime assertion checking) of thread-safety properties as well as functional properties in a concurrent environment without requiring atomicity to be established a priori. It is also the first to propose a complete solution to the problem of interference without limiting the use of concurrency constructs, thus allowing concurrent programs in Java-like languages to be completely specified.

5 Case Study–Specification of an Industrial Concurrent System

This section describes the application of the proposed constructs in the specification of a portion of an industrial system using JML and analyzes their suitability in terms of the behaviours they can specify and the ones they cannot. We take the approach of specifying the behaviour presented by the code without introducing any changes to improve its specifiability.

The target system is the Service Activation Engine (SAE) component of the Session Resource Controller product line of Juniper Networks. It is basically a platform to design and deploy value-added services in an Internet Protocol network. It does so by converting service definitions specified as an abstract set of traffic controlling policies for a particular subscriber into device specific policies in the context of the interface the subscriber uses to connect to the network. The SAE currently supports various devices, and we focus here on the subsystem that interfaces with Juniper's JUNOSe routers. This subsystem, called router driver, is responsible for responding to asynchronous notifications

from the router regarding the state of each subscriber interface and managing traffic policies for each interface. Due to the large number of subscribers a router supports, these requests are processed concurrently to maximize system performance. The router driver is responsible for the translation task above, the low-level communication with the router and ensuring correctness in the presence of concurrent processing. It does so by implementing a transactional infrastructure to guarantee ACID (Atomicity, Consistency, Isolation, and Durability) properties of transactions. This system is capable of managing approximately 520,000 active subscribers connected to multiple JUNOSe routers. This amounts to executing approximately 1,500 transactions per second. The complex functionality of this system allows the use of complex functional specification constructs, and its high degree of concurrency with varied and intricate concurrency control patterns allows for all proposed constructs to be explored.

The router driver subsystem is composed of 54 classes and interfaces (33509 LOC), all of which are used in a concurrent environment. Of these, 34 present concurrent behaviour. Table 1 summarizes the usage of the thread-safety constructs we propose. The first column provides a count of the methods considered. The second column lists the number of methods subject to specification inheritance. The third column provides a count of the methods that use the **`requires_/ensures_thread_safe`** constructs. The fourth column counts the use of locking predicates. All columns show both the total number of methods that require those constructs and the total number of methods that were successfully specified with those constructs. We consider a specification to be successful if we are able to specify the method concurrent behaviour. A method can require several of those constructs. The first line lists the absolute numbers and the second one shows percentages (the ones in the 'Succ' column are relative to the adjacent 'Total' column to the left).

Table 1. Statistics on the usage of the proposed thread-safety constructs for the case study.

	Number of methods		Thread-safety spec inheritance		Thread-safe uses		Lock predicate uses	
	Total	Succ	Total	Succ	Total	Succ	Total	Succ
Total	307	279	104	102	83	56	42	42
Percentage	100	90.9	33.9	98.1	27.0	67.5	13.7	100

Let us highlight some of the most important results in Table 1. First, 33.9% of all methods require the specification of inheritance with concurrency, demonstrating the significance of representing this situation properly. It also shows that our constructs are able to specify 98.1% of these cases correctly. Second, 27.0% of all methods make use of thread-safety clauses independently of specification inheritance, of which 67.5% are correctly specified by our constructs. Third, this table shows that all cases involving the use of lock predicates were correctly specified by our constructs.

The cases that could not be specified (i.e., 29 methods), including both specification inheritance and thread-safe uses, were due to the fact that objects that perform concurrency control internally (concurrent objects) can never be entirely thread-safe but they can be *piecewise thread-safe*. An object is piecewise thread-safe if it can be partitioned into groups of methods or fields that do not interfere within a group and for which concurrency control across groups is taken care of internally by the object. A producer-consumer scenario in which up to one producer and one consumer threads are allowed to execute disjoint sets of operations on an object is one example. It is currently impossible to specify such concurrent behaviour, either with JML or our extensions. We are working on extending the thread-safety constructs to accommodate this.

Table 2 summarizes the uses of safepoints. The second column displays the total number of methods that successfully use safepoints. Table 2 shows that 32.9% of all methods required the use of safepoints to be correctly specified and that our proposed constructs succeeded in specifying 90.1% of these cases. The major limiting factor was the fact that it is not always possible to have safepoints in the method body due to the requirement of not allowing observable statements to happen before (after) a precondition (postcondition) safepoint. In such cases, only limited predicates can be safely evaluated.

Table 2. Statistics regarding the use of safepoints.

	Number of methods	Safepoints uses	
		Total	Success
Total	307	101	91
Percentage	100	32.9	90.1

This case study shows that our proposed constructs are not only essential to the proper specification of concurrent programs but that they are also capable of specifying most behaviour. The thread safety constructs are not able to specify what we call piecewise thread-safe behaviour of objects, which amounts to 15.5% of the eligible methods (187). This is a significant limitation which we are working to overcome. The use of safepoints to specify properties of a concurrent system proved to be not only essential but also applicable to the vast majority of cases. Most cases that could not be specified could have been so by reorganizing the method's code to allow for the placement of safepoints, thus confirming that safepoints are capable of specifying concurrent object-oriented programs. Some failed to be specified due to the inherent non-determinism of concurrent systems. For instance, it is not always possible to guarantee that after taking an

element from a concurrently modified queue it will contain one element less than it had prior to executing this operation. This is a limitation of the (method) implementation, not the DbC technique we propose, since one can trivially eliminate all concurrency control issues by externally acquiring all necessary locks prior to executing a method to guarantee its sequential execution.

6 Conclusion

Applying Design by Contract to concurrent software poses several challenges. We tackle interference with the introduction of safepoints. We define their syntax and semantics in the context of a concurrent method. Based on this concept, we also derive minimum thread-safety requirements for a method to be interference-free.

We address thread-safety specification by separating the concurrent aspect of a contract, which houses clauses to specify thread-safety properties, from the functional aspect thus maintaining the usual notions of behavioural subtyping for the specification of functional properties.

Freedom from interference allows the use of sequential contracts in a concurrent environment. Not every sequential contract can be expressed as a correct concurrent contract, though. However, this is not a limitation of our technique but of the method being specified.

We implemented our proposed constructs on the JML toolset, including the Runtime Assertion Checker, although this is not discussed in this paper due to size constraints. We validated these constructs with an industrial case study. We identified some limitations but we were nevertheless able to specify complex behaviours, both functional and concurrent, that could not be specified with the current JML constructs.

Our next step is to continue the work of Briand et al. [6] to determine the effect of the complexity of contracts used as test oracles in the detection of faults with an emphasis on concurrent systems.

Acknowledgements. This research was supported by Juniper Networks, Inc. Lionel Briand and Yvan Labiche are supported by NSERC. Thanks to the JML community for the many fruitful discussions and the great software you produced. Special thanks to Gary Leavens and Clément Hurlin for the extensive discussions about JML and concurrency, which helped clarify fundamental points in this work.

7 References

[1] Agrawal R., Sasturkar A., Wang L. and Stoller S. D., "Optimized Run-Time Race Detection And Atomicity Checking Using Partial Discovered Types," *Proc. ASE*, pp. 233-242, 2005.

[2] America P., "Inheritance and Subtyping in a Parallel Object-Oriented Language," *Proc. ECOOP*, LNCS 276, pp. 234-242, 1987.

[3] Araujo W., Briand L. and Labiche Y., "Concurrent Contracts for Java in JML," Carleton University, Technical Report SCE-07-11, http://squall.sce.carleton.ca/, 2007.

[4] Arnold K., Gosling J. and Holmes D., *The Java Programming Language*, Addison-Wesley, 2000.

[5] Arslan V., Eugster P., Nienaltowski P. and Vaucouleur S., "SCOOP - concurrency made easy," in B. Meyer, A. Schiper, and J. Kohlas, Eds., *Dependable Systems: Software, Computing, Networks*, vol. LNCS 4028, pp. 82-102, 2006.

[6] Briand L. C., Labiche Y. and Sun H., "Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code," *SPE*, 33 (7), pp. 637-672, 2003.

[7] Cheon Y. and Leavens G. T., "A Runtime Assertion Checker for the Java Modeling Language," *Proc. Software Engineering Research and Practice*, pp. 322-328, 2002.

[8] Cheon Y., Leavens G. T., Sitaraman M. and Edwards S., "Model Variables: Cleanly supporting Abstraction in Design By Contract," *SPE*, 35 (6), pp. 583-599, 2005.

[9] Dhara K. and Leavens G. T., "Forcing Behavioural Subtyping Through Specification Inheritance," *Proc. ICSE*, pp. 258-267, 1996.

[10] Flanagan C. and Freund S. N., "Atomizer: a dynamic atomicity checker for multithreaded programs," *Proc. ACM SIGPLAN/SIGACT POPL*, pp. 256-267, 2004.

[11] Jacobs B., Leino R. M., Piessens F. and Schulte W., "Safe concurrency for aggregate objects with invariants," *Proc. ICSE*, pp. 137-147, 2005.

[12] Jacobs B., Leino R. M. and Schulte W., "Verification of Multithreaded Object-Oriented Programs with Invariants," *Proc. ACM Specification and Verification of Component Based Systems*, pp. 2-9, 2004.

[13] Leavens G. T., Baker A. L. and Ruby C., "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 31 (3), pp. 1-38, 2006.

[14] Leavens G. T. and Müller P., "Information Hiding and Visibility in Interface Specifications," Department of Computer Science, Iowa State University, Technical Report 06-28, 2006.

[15] Leavens G. T., Poll E., Clifton C., Cheon Y., Ruby C., Cok D., Müller P. and Kiniry J., "JML Reference Manual," Department of Computer Science, Iowa State University, http://www.jmlspecs.org, 2007.

[16] Liskov B. H. and Wing J. M., "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16 (6), pp. 1811-1841, 1994.

[17] Meyer B., "Design by Contracts," *IEEE Computer*, vol. 25 (10), pp. 40-52, 1992.

[18] Nienaltowski P. and Meyer B., "Contracts for concurrency," *Proc. International Symposium on Concurrency, Real-Time and Distribution in Eiffel-like Languages*, 2006.

[19] Rodríguez E., Dwyer M., Flanagan C., Hatcliff J., Leavens G. T. and Robby, "Extending JML for Modular Specification and Verification of Multi-threaded Programs," *Proc. ECOOP*, LNCS 3586, pp. 551-576, 2005.