Concurrent Contracts for Java

Wladimir Araujo\textsuperscript{1}, Lionel C. Briand\textsuperscript{2,3}, Yvan Labiche\textsuperscript{3}

\textsuperscript{1} Juniper Networks
700 Silver Seven Road
Ottawa, ON, K2V 1C3, Canada
waraujo@juniper.net

\textsuperscript{2} Simula Research Laboratory and University of Oslo
P.O. Box 134, Lysaker
1325 Norway
briand@simula.no

\textsuperscript{3} 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

1 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 and verification of locking related properties. The third 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.

2 INTRODUCTION

Including specifications of program behaviour together with the source code is not a new idea. Design-by-Contract (DbC) [22] is one of the more elaborate software development methodologies that put such idea in practice, with Eiffel being a well-known example of a programming language that supports it.
The Java Programming Language [29] does not provide native support to DbC. It only provides basic support for assertions through the `assert` keyword, which simply causes an exception to be thrown in case a given Boolean expression evaluates to false. We chose the Java Modeling Language (JML) [8][9] as the specification language used to write contracts. It allows the specification of properties from simple assertions (lightweight properties) about pointer null-ness to complete functional correctness of program components (strong properties). JML is a behavioural interface specification language with which one can specify the syntactic and behavioural interface of a portion of Java code. It also includes notations for pre- and postconditions, and invariants, thus providing support for the Design-by-Contract paradigm. JML has a Java-like syntax and specifications can even perform method calls in assertions. It also provides a rich set of model classes (i.e. classes that can only be used in specifications) that can be used to construct rich abstract descriptions of program behaviour, such as data structure model classes, which can be used to model abstract properties of concrete data structures in a concise way.

The idea behind DbC is that a method defines a contract stating the requirements a client needs to fulfil to use a particular method, the precondition, and the properties such method ensures after its execution, the postcondition. A contract says nothing about the method behaviour in case the requirements are not respected. The use of pre- and postconditions to specify software was introduced by Hoare [26] in the context of formal verification. The logic used to reason about program correctness became known as Hoare logic. DbC extends this idea by making contracts executable [25]. 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

---

1 JML allows one to specify loop variants and invariants, internal method properties, external method properties and properties of types (classes and interfaces).

2 Meyer originally named them class invariants but we prefer the term object invariant since it is an invariant about an object [35].
before a visible method execution, and just after a visible method execution. In the context of JML, the only types of methods that are not considered visible are the ones marked as helpers. Such methods do not implicitly rely on the class invariant and are also not obliged to establish it. They are useful to establish intermediate states as factored out routines used by other, possibly visible, methods.

Behavioural subtyping \[28\][36][37][38][39] is also an integral part of DbC. A subtype automatically inherits the specification (contracts and invariants) from its super-types \[40\][41][42]. The effective precondition of a method is the disjunction of all the inherited preconditions and the method’s declared precondition. The effective postcondition is the conjunction of all inherited postconditions and the method’s declared postcondition. 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). JML implements such mechanism through specification inheritance.

The JML toolset comes with a compiler that translates specifications into runtime assertion checking (RAC) code producing Java classes augmented with executable assertions. The process of adding RAC code to a Java class is called instrumentation. The resulting class is called the instrumented class. The JML compiler \[10\] produces RAC code that enforces behavioural subtyping, i.e. RAC code for all applicable invariants and preconditions is executed upon entering a method, and RAC code for all applicable postconditions and invariants is executed upon exiting a method.

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. Interference is present even on correct programs with respect to concurrency control. Basically, interference with respect to the precondition happens because assertion checking code is evaluated at a point in time after which other threads are allowed to modify the objects referenced in such assertions but prior to the point in which these objects are accessed by the method in question. This causes RAC code to report errors for correct methods and vice-versa. The problem is analogous with respect to postconditions.
and invariants. 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 and verification of locking related properties. It is common practice to have methods in a concurrent class (informally) specify which locks the client is required to hold prior to executing such method. It is also common practice to specify which locks the method in question is going to (potentially) acquire during its execution. In some cases, it is also useful to specify the order in which certain locks must be acquired to avoid deadlocks. The use of locking policies is a common deadlock avoidance technique [47]. JML already has constructs to deal with the first two cases (the construct addressing the first case presents inheritance issues, though). We introduce the concept of a lock order clause, which is a specialized construct to specify only lock acquisition order expressions. Its semantics makes the use of quantifiers unnecessary and also eliminates the soundness issues with the current approach.

The third challenge is the specification of thread-safety properties in the presence of inheritance. To our knowledge, locking requirements have so far been associated with preconditions. This causes two problems. It implicitly associates locking requirements with behavioural subtyping: satisfying them becomes optional since inherited preconditions are disjoined with the local one, and sub-types with potentially different concurrency control strategies are forced to inherit possibly meaningless locking requirements. The same problem happens with thread-safety predicates. These properties only make sense in the context of the target type, not its hierarchy. We solve this problem by detaching these properties from preconditions while considering interference issues. This is another contribution of this work.

The following section describes these three challenges in detail, presenting examples and identifying the issues in specifying various properties and behaviours related to concurrency. The next section presents the solutions to these problems. It is the section that describes our contributions. Related work is discussed in the subsequent section. The next section describes the specification of an industrial system as an application of the
techniques we developed. We conclude with a summary of our contributions and future work.

3 CONTRACTS AND CONCURRENCY

This chapter presents the problems associated with using contracts to specify the behaviour of concurrent programs and to generate runtime assertion checking code from such contracts. Java and JML are used in all examples thorough this document. Although we do not describe JML in detail we present a brief description of the relevant constructs just enough for the understanding of the intent of a particular contract and the challenges in using it in a concurrent environment. We also generalize these problems to Java-like object-oriented languages and specification languages through the use of UML diagrams to describe the program under study and OCL to describe the contracts, respectively. We only require from the specification language that it be able to specify formulae for preconditions, postconditions and invariants, that it supports abstraction through model fields, and that it supports specification inheritance.

The problems presented in this chapter cover different aspects of the verification of concurrent contracts. Some of these problems have been originally presented in [7] or at least restated in the context of Java and JML. We find such solutions, although interesting, inadequate to the complete scope of an object-oriented language like Java, with intricate concurrency control mechanisms.

We start by discussing the problems of internal and external interference, originally presented in [7], or at least restated in the context of Java and JML. We defer the presentation of our solution to these problems until the next chapter since it is necessary to have a complete view of the problems it solves before evaluating its effectiveness.

We then move to the problem of specifying and verifying concurrent contracts in the presence of specification inheritance. Specification inheritance is the mechanism used by JML to achieve behavioural subtyping. This is an original contribution of this work.
We then continue to the problem of specifying and verifying blocking behaviour. A simple statement of the problem of specifying such behaviour was made in [7] as an adaptation of [16] to Java and JML. Such statement implies that the method being specified is atomic. We use the same constructs to specify blocking behaviour but we do not make any assumptions regarding atomicity. Furthermore, we present the problem taking inheritance into account, which has never been done before.

We tackle locking policies next. Although there are mechanisms to specify locking policies today [31], the current solution is not amenable to runtime assertion checking. We present such shortcomings. The solution is, again, deferred to the next chapter.

We finally address the problem of specifying and verifying thread-safe behaviour. The notion of thread-safety is borrowed from [7]. We restate the problem with a focus on runtime assertion checking (RAC) and show why the current approaches are insufficient. Furthermore, we bring specification inheritance into the picture and present several issues with combining thread-safety properties with functional properties. This is another contribution of our work.

### 3.1 The Problem of Internal Interference

Two threads interfere when one 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. The example below (from [7]) illustrates the situation quite well.
public class LinkedQueue {
    protected /*@ spec_public non_null @*/ LinkedNode head;
    protected /*@ spec_public non_null @*/ LinkedNode last;
    //@ public invariant head.value == null;

    //@ public normal_behavior
    @ requires head == last;
    @ assignable \nothing;
    @ ensures \result == null;
    @ also public normal_behavior
    @ requires head != last;
    @ assignable head, head.next.value;
    @ ensures head == \old(head.next) &&
    @ \result == \old(head.next.value);
    @*/
    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;
        }
    }
}

Listing 1: JML specification for the method extract of the LinkedQueue class.

Before delving into this example, the notion of a method’s pre-state and post-state must be presented. “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” ([8], p. 8).

The method specification in Listing 1 simply tells that if the field head references the same object as last, the method is not allowed to modify anything and that it will return null. It also tells that if that is not the case, the field head and the value member of the successor of head are the only locations that can be modified, and that head will reference the node that was its successor in the method’s pre-state and that it will return the value field such node had in the pre-state. Or, in a more colloquial language, the
head of the list will move to the next element and it will return the value of what used to be the first element of the list.

The preconditions of a method (i.e. the expression in the \texttt{requires} clause) are evaluated in the method’s pre-state. Also, the expressions provided as arguments to the \texttt{old} operator in the method postcondition are evaluated in the pre-state of the method. The method postconditions are evaluated in the method’s post-state.

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

This example can also be visualized on the UML interaction diagram on Figure 1. We use the notation from the UML Profile for Schedulability, Performance and Time to depict concurrent execution by placing two activations side-by-side. In this example, it means that methods \texttt{extract()} and \texttt{insert()} of \texttt{LinkedQueue} are executed concurrently. The self message “eval precondition” flags the evaluation of the precondition and invariant predicates in the pre-state. At that point, \texttt{thread1} observes that \texttt{head==null} and thus selects the specification case in which \texttt{\result==null} is the expected postcondition. The method execution proper begins once this self-action returns. We label the points in which \texttt{thread1} is pre-empted and resumed using notes. Notice that \texttt{extract()} executes completely while \texttt{thread1} is sleeping. Once \texttt{thread1} resumes, it proceeds to return an element from the head of the queue (the one just inserted) therefore causing \texttt{\result} to not be \texttt{null}. The last step of the method execution is to execute the self-action “eval postcondition”. At this point it
recalls the selected postcondition in the pre-state and evaluates it. Although the behaviour of both methods is correct, the postcondition is not satisfied, producing a false negative.

![Sequence diagram illustrating the problem of internal interference.](image)

**Figure 1:** Sequence diagram illustrating the problem of internal interference.

The behaviour just described is called internal interference, when another thread affects the current thread’s execution of a method by making changes observable by such method once it started executing [7]. This problem is not specific to Java or JML. Any object-oriented language in which the sequence diagram above is realizable and provides support for DbC via runtime assertion checking (RAC) is prone to the same issues. 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.

### 3.2 The Problem of External Interference

Another type of interference happens when a thread makes observable changes between a method call and a method’s entry, or between a method’s exit and the caller’s resumption.
This is called external interference. Such behaviour can be seen in the following example.

![Sequence diagram](image)

**Figure 2: Sequence diagram depicting the problem of external interference.**

Two threads accessing the same instance of LinkedQueue, one executing the method `extract()` from Listing 1 (thread 1) and another executing the method `isEmpty()` from Listing 2 (thread 2), interfere with each other if thread 2 executes `isEmpty()` completely (i.e. it returns) but is pre-empted before resuming to the client (i.e. before the method’s post-state) and thread 1 executes `extract()` completely. If this queue has only one element before both threads started executing, `isEmpty()` returns `false`, since it perceives the element in the queue. After `extract()` executes, the queue becomes empty. Once thread 2 resumes, it evaluates the postcondition of `isEmpty()` in its post-state. At this point, the queue is empty and the postcondition formula evaluates `true`. 
This example can be seen on Figure 2. We represent the “instantaneous” execution of `extract()` by not drawing a return arrow. The problem in this case is localized on the evaluation of the postcondition. Once `isEmpty()` returns `false`, thread2 is preempted before executing the assertion checking code in the post-state and `extract()` executes completely, which results in setting `head.next` to `null`. Once thread2 resumes, the postcondition evaluation code perceives the value of `head.next` set by `extract()`, not the one at the point `isEmpty()` returned. This causes the postcondition to evaluate `false`, producing a false negative.

External interference breaks the Hoare-style reasoning by making observable changes between a method execution and the evaluation of the associated assertion formula, either for pre or postconditions.

```java
/*@ normal_behavior
@ ensures \result <=> head.next == null;
@*/

public boolean isEmpty()
{
    synchronized (head)
    {
        return head.next == null;
    }
}
```

Listing 2: JML specification of method `isEmpty()` of class `LinkedQueue`.

A closer look at the contract in Listing 2 tells that the reason why external interference is a problem for this particular case is because the ensures clause (the postcondition) makes reference to the internal state of the object, which can be concurrently modified by other threads. Once the current thread reaches the post-state of method `isEmpty()` it no longer has a lock on the relevant object for this contract, namely the object pointed to by the field `head`. One must also note that the precondition for this method is `true`, i.e. the client has no obligations. A similar case happens on Listing 1 for the preconditions. They reference the private field `head` and `last` and their fields through the `\old` expressions in the postcondition clauses. Although located in the `ensures` clause, the values of the `\old` expressions are computed in the method pre-state to be later referenced in the post-state while evaluating the postconditions. One should also note that the effective
precondition of this method is \texttt{true}. The specification cases (conjoined by the keyword also) simply specify the different behaviours of the method depending on the queue being empty or not. They do not impose any obligations onto the client, which is allowed to call \texttt{extract()} whether or not the queue is empty.

The issue of information hiding and modular reasoning for concurrent contracts is left for a later section. At this point, the focus is on the fact that threads interleaving cause problems to concurrent contracts because the evaluation of such contracts leads to access to unprotected data in the method pre- and post-states. Another way of looking at it is to notice that the time the method observes these values (e.g. in an \texttt{if} statement) is different from the time the contract observes them. In other words, “stuff” happens between the point a thread observes a variable on a contract and the same thread observes the same variable in the method, and what happens is not under the control of the observer thread. Our solution to this problem (presented on the next chapter) is to make sure the contract and the method observe the same values at the same time.

3.3 Specification Inheritance

Specifications can be inherited from interfaces and super-classes. The example in this section makes use of an interface to illustrate the issues with data abstraction and concurrency. The issues are the same in the case of a super-class. Specification inheritance is a form of implementing behavioural subtyping [40].

Listing 3 shows the specification of the Channel interface. It declares two model fields. A model field (a field with the \texttt{model} modifier) is a field that does not have to be implemented but can be used in a specification as any other field. The “value” of a model field can be defined in implementing classes or, in the case of being declared on a class, by concrete fields of such class or its subclasses. Both fields in this example are marked \texttt{instance}, meaning that they are fields of the object implementing the interface instead of static fields of the interface. Java only allows static final fields on interfaces. JML allows both static and instance fields on interfaces. They can only be used inside specifications, obviously.
/**
 * A channel can be used to send messages between peers. Messages
 * sent on a channel are delivered in the order they have been sent.
 * Sending and receiving messages are independent tasks for the same
 * peer, i.e. one can send and receive messages without concerns if
 * the remote peer has read previously sent messages.
 */

public interface Channel
{
    /**
     * Receives the next message from the remote peer.
     * @return the next message
     * @throws NotConnectedException if the channel is not connected
     */
    public Message receive() throws NotConnectedException;
}

Listing 3: JML specification of interface Channel. Only the receive method and
relevant model fields are shown for simplicity.

Interface Channel can be implemented with the help of a Pipe (from Listing 5), as
shown by class PipedChannel in Listing 4. The represents clause maps the value
of a model field to an expression based on concrete fields of the class. The model field
connected obtains its value from the concrete fields closed and remoteClosed
according to the Boolean expression !closed && !remoteClosed. The value for
the model field nPending comes from the input pipe in, via its getSize() method.
public class PipedChannel implements Channel {

    /**
     * The pipes.
     */
    protected final Pipe in, out;

    private boolean closed = false; // @ in connected;

    private boolean remoteClosed = false; // @ in connected;

    /*@
    private represents connected <= !closed && !remoteClosed;
    private represents nPending <= in.getSize();
    */

    public Message receive() throws NotConnectedException {
        if (closed)
            throw new NotConnectedException();

        if (in.isEmpty())
            synchronized (this) {
                if (remoteClosed)
                    { 
                        closed = true;
                        throw new NotConnectedException();
                    }
            }

            return null;
        }

        return in.take();
    }
}

Listing 4: JML specification of class PipedChannel. Only relevant methods, fields and specifications are shown for simplicity.

Since method receive() in Listing 4 does not declare any specifications, it inherits the parent’s specification without any changes. The inherited normal specification (the one that specifies the conditions for the method to return normally) in Listing 3 simply says that receive() will return any Message objects the channel contains even if it has already been closed or null if it is empty. The exceptional specification states that receive() will throw a NotConnectedException if the channel has been closed and it is empty.
The problem with 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 on 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.

The issue here is more subtle than on the previous cases. Model fields are used to specify behaviour without giving out implementation details. They can be seen as mathematical entities on which it is safe to reason about the behaviour of the object. Model fields do not have storage, i.e. their values are derived from the container object’s state, not through direct assignment, which is forbidden. In the examples above, one can see that the value for \texttt{nPending} comes from the evaluation of the \texttt{getSize()} method of the field \texttt{in}. One can think of a model field as a read-only field that is modified outside of the specification control. Apart from just allowing abstract modelling [12][46], model fields play a vital role in information hiding, modular reasoning and behavioural subtyping [27]. Thus, in comparison with the examples on previous sections, the contracts on this example are not accessing internal data directly; they are making use of the available modelling facilities to best describe the behaviour of the object in question.

A solution to the issue of contracts for concurrent objects must accommodate the use of model fields. Current solutions do not even acknowledge the existence of a problem. One approach to the problem is to require the client to guarantee thread-safety of all fields referenced in method specifications [43][44]. Apart from other issues related to concurrency control we discuss in section 5, this solution cannot be applied to model fields at all. Clients do not have visibility on how the provider realizes model fields and therefore cannot know which locks to acquire. Furthermore, if the client only sees the interface, it might not even know that there are concurrency control issues. The problem presented by verifying abstract specifications in a concurrent environment is then that, in general, the client cannot be required to perform additional concurrency control simply to
guarantee thread-safe access to fields present in such specifications because they do not present any concurrency control related predicates or because the specification refers to model fields.

3.4 Blocking Behaviour

Java implements blocking behaviour through the use of the *wait()* primitive on a monitor on which the executing thread\(^3\) holds a lock [29]. The use of such feature usually follows the pattern of a guard condition that must be satisfied for the method to proceed executing. While such condition is not satisfied, the method blocks execution of the current thread, releasing its lock on the target monitor object. Upon a state change, other threads notify the waiting thread(s) via the *notify()* or *notifyAll()* primitives of the monitor lock, upon which time a waiting thread wakes up, acquires the lock on the monitor object and rechecks the guard condition. If it is satisfied, the method proceeds executing. If not, it blocks again.

This general pattern derives from monitors according to Hoare [45]. A monitor contains private data, condition variables and procedures to acquire and release it. A condition variable offers the operations *wait()* which causes the executing thread to block and relinquish its lock on the monitor, and *signal()* which causes a blocked thread to wake up and start executing as soon as it obtains a lock on the monitor. The monitor acquisition operation checks a wait condition (a predicate based on the monitor’s private data) to decide if it should wait or if it should proceed with the acquisition. The wait condition must be satisfied for the monitor acquisition to proceed. Once satisfied, the wait condition can (and likely will) be invalidated by the acquisition procedure. Therefore, a wait condition is only guaranteed to be valid at a particular point inside the acquisition procedure. After such point, the acquisition procedure executes without waiting. Since a monitor is equivalent to a semaphore (i.e. one primitive can be used to implement the other) our presentation based on Java is valid for any other object-oriented language that uses either monitors or semaphores as the mutual exclusion primitive.

\(^3\) In Java terminology, the executing thread is also known as the *current thread*. These terms are used interchangeably throughout this text.
JML makes use of the `when` clause to specify wait conditions [7]. The `when` clause specifies a predicate that must be satisfied at a particular point of a method execution after which the executing thread is not allowed to block. Such point is called the commit point (from the notion of commit atomicity) and is marked with the JML label `commit` inside the method. Furthermore, the specified predicate is only required to hold at the commit point; nothing is said regarding its validity afterwards. Other threads or even the current thread are allowed to invalidate it after the commit point.

Listing 5 shows an example of the use of the `when` clause. Method `get()` will block until the pipe is not empty, after which it will return a non-null message. If the pipe is closed, however, it will return `null`. This contract is quite simple but nevertheless presents the problem of external interference due to thread interleaving with respect to the `closed` field.
/**
 * A pipe allowing messages to be put on one side and taken on
 * the other. It can be used by multiple threads.
 */

class Pipe {
    private final /*@ monitored@*/ LinkedList buf = new LinkedList();
    private /*@ monitored spec_public @*/ boolean closed = false;

    public synchronized void put(Message aM) {
        if (buf.isEmpty())
            notify();

        buf.addLast(aM);
    }

    /*@
    public normal_behaviour
    requires closed;
    ensures \result == null;
    also
    public normal_behaviour
    requires !closed;
    when !isEmpty();
    ensures \result != null;
    */
    public synchronized Message get() throws InterruptedException {
        while (buf.isEmpty() && !closed)
            wait();

        //@ commit:
        if (closed)
            return null;

        return (Message) buf.removeFirst();
    }

    public synchronized /*@ pure @*/ boolean isEmpty() {
        return buf.isEmpty();
    }

    public synchronized /*@ pure @*/ int getSize() {
        return buf.size();
    }

    synchronized void close() {
        closed = true;
        notify();
    }
}

Listing 5: The class Pipe and its JML specification.
The interesting point of this example is that there are interference issues regarding the evaluation of the wait condition. The commit point demarcates the location where the predicate can be safely evaluated. However, if external interference caused `closed` to be seen as `false` in the pre-state but `true` inside the method, the wrong wait condition would be evaluated (a missing `when` clause has a default value of `true`)\(^4\) since the second specification case would be selected instead of the first.

The issue of this example is, again, the fact that specification cases observe internal state without acquiring the proper locks. In this particular example it is even clearer than on the previous one. The field `closed` has the `monitored` modifier, which states that accesses to it must be performed while holding a monitor lock on the enclosing object. Thus, for the precondition to refer to it while on the pre-state would already constitute a violation from the perspective of runtime assertion checking.

The problem of internal interference is implicitly dealt with in [7] since methods are expected to be atomic. Atomicity, however, does not guarantee absence of external interference. It only guarantees that the method can be treated as if it was executing sequentially from a reasoning point of view. To implement runtime assertion checking of such method requires the same measures required for a non-atomic method.

Combining blocking behaviour with specification inheritance yields a unique problem. Class `BlockingChannel2`, in Listing 6, illustrates this situation. It extends class `PipedChannel` (from Listing 4) to add blocking behaviour to the `receive()` method. The JML specification also shows an additional specification case for this method. The keyword `also` in the beginning of a method specification denotes inheritance\(^5\): the resulting method specification is the combination of the one seen on this

\(^4\) This is the case in this example due to the use of the `normal behaviour` keyword, which marks a specification as heavyweight. Heavyweight specifications have well defined default values for omitted clauses, as opposed to lightweight specifications, in which omitted clauses have a default value of `not_specified`. The default values for heavyweight specifications including more details about them can be found in [9].

\(^5\) An extending method specification must always start with the keyword `also`. It is not an option for a method specification to not inherit its supertypes’ specifications.
class and the one on the Channel interface (the superclass PipedChannel does not add anything, in this example).

```java
/**
 * A channel that blocks during receive if there are no available messages.
 */
class BlockingChannel2 extends PipedChannel {

/**
 * Receives the next message from the remote peer.
 * Blocks until there is a message available.
 * @return the next message
 * @throws NotConnectedException if the channel is not connected
 */
//@
also
public normal_behaviour
  requires nPending > 0 || (nPending == 0 && connected);
  when nPending > 0;
  ensures \result != null;
//@

public Message receive() throws NotConnectedException {
  try {
    synchronized (in) {
      while (in.isEmpty() && !disconnected)
        in.wait();
      
      if (in.isEmpty())
        throw new NotConnectedException();
      //@commit:
      return super.receive();
    }
  }
  catch (InterruptedException x) {
    throw new AssertionError();
  }
}
}
```

Listing 6: JML specification of class BlockingChannel2. Irrelevant methods and fields are omitted for clarity.

JML enforces behavioural subtyping. The resulting method specification works as if this method’s specifications from all super-types were copied to the current type and joined by the keyword also into specification cases. Specification cases are equivalent to a single specification case in which all preconditions are disjoined and the postcondition is
the conjunction of all the implications of each case precondition to its postcondition. Listing 7 shows two equivalent method specifications: the first, for method \texttt{m()} with specification cases and the second, for method \texttt{m2()}, without. The complete desugaring process for all JML clauses and constructs can be found in [30]\textsuperscript{6}.

```java
/*@
  public behaviour
  requires P;
  ensures Q;
  when R;
  also
  requires P1;
  ensures Q1;
  when R1;
*/
public void m();

/*@
  public behaviour
  requires P || P1;
  ensures (\old(P) ==> Q) && (\old(P1) ==> Q1);
  when (\old(P) ==> R) && (\old(P1) ==> R1);
*/
public void m2();
```

Listing 7: Example of desugaring of method specifications. The two method specifications are equivalent. P, P1, Q, and Q1 represent JML predicates.

The resulting specification for \texttt{BlockingChannel2.receive()} contains the problems discussed in the previous section. It also contains the problems due to external interference discussed in section 3.4. It contains yet another problem resulting from the addition of blocking behaviour to a sub-type’s method: the parent’s specification becomes invalid. Looking at just the local specification of \texttt{receive()} in Listing 6, one should note that the only guarantee this method offers is that it will return a non-null value. This is due to the blocking behaviour. The \texttt{when} clause guarantees that the method will block until there are messages available to be received but it does not say how many. While a client blocks waiting for a message to arrive, any number of messages might be put into the pipe. This breaks the contract on the \texttt{Channel} interface,

\textsuperscript{6} The JML literature uses the term effective specification to designate the result of the desugaring process. The term effective specification has a slightly different meaning in this report. It means the desugared specification in which all the predicates have been minimized.
which specifies that the number of pending messages after the method executes will be one less than what it had prior to its execution.

Simply put, adding blocking behaviour to a method likely breaks its inherited specifications, i.e. it causes it not to be a behavioural subtype. This is not the whole truth, though. Setting the issue of interference aside for a moment, one can notice that if the thread does not actually block during the method execution, the contracts can be honoured. `BlockingChannel2.receive()` is an example of this case. Since the thread holds a lock on the field `in` while calling the superclass’ implementation of the method, there is no possibility of interference, and thus all contracts are satisfied.

This apparent problem of the JML semantics (or of behavioural subtyping in general) needs further discussion. The issue here is caused by the inherent nature of blocking. Once a thread blocks, it releases its lock on the monitor on which it blocks to allow other threads access to it so they can modify the state up to the point the wait condition can be satisfied and the blocking thread can proceed executing. Looking at the contracts for `receive()` in Listing 3 and Listing 6 one should note that both preconditions for which the method returns normally are identical. One should also note that both contracts are satisfied if the wait condition `nPending > 0` is satisfied at precondition time. The problem occurs in the case the wait condition is not satisfied at precondition time, i.e. if the pipe is empty. This causes the current thread to block until it is satisfied. In a sense, this particular specification case is negating its own precondition and by doing so it is satisfying the precondition of the other specification case. But since `\old(nPending)` was computed at precondition time instead of reflecting the value of `nPending` at the commit point, the postcondition fails to be satisfied. The problem here is not with behavioural subtyping but with the contracts themselves: the wait condition refers to a predicate present in the precondition that may not have been satisfied at precondition evaluation time, there is more than one specification case with satisfying preconditions, and the satisfaction of the wait condition implies the satisfaction of the other case’s postcondition if the value of `\old` variables were recomputed at the commit point. This situation causes the subtype to not be a behavioural subtype because the implementation will not be able to satisfy the specification in all possible interleavings.
3.5 Locking Policies

The specification of locking policies is one of the strategies to guarantee deadlock freedom in concurrent software [47]. Locking policies can control many aspects of the locking acquisition and release process. In this work we restrict ourselves to the specification of lock acquisition order (from the language standpoint). Lock ordering works as a building block to specifying more complex policies.

The intent of specifying lock acquisition order policies is to define conditions regarding the order in which multiple threads can safely acquire locks. By abiding to such conditions, threads are guaranteed to never deadlock, assuming that such conditions are specified correctly. Typical policies one wants to specify are “no locks should be acquired after acquiring a lock on an instance of this class” or “acquiring a lock on a tree node requires the acquisition of a lock on its parent”. The aim of this section is to describe the issues with the current approach of specifying such conditions in the context of runtime assertion checking.

JML overloads the operators < and <= when applied to objects to evaluate the order the current thread acquired the locks given as arguments. A lock object l1 is smaller than another l2 if the current thread acquired l1 before having acquired l2. JML also provides the \texttt{max} expression, which returns the largest of a set of objects according to the above definition of <.

The current way to specify lock ordering policies uses the axiom statement (see section 2.7.2 of [31]). Axioms are predicates that can be specified inside a class declaration. They can be used by theorem provers for correctness verification. The RAC ignores axioms [10]. The use of axioms to specify lock ordering properties presents several disadvantages. First, they are not part of class invariants and, thus, are not part of a contract. Although the semantics of axioms states that it should be valid at the beginning of every method execution, it does not say anything with respect to its validity at the end of such execution. Hence, it cannot be considered a class invariant. Second, axioms accept a general predicate as an argument. Although flexible, this can lead to several
reasoning complications that are not considered worthwhile to tackle (e.g. having to use theorem proving to analyze the predicates). Third, it is possible, although infrequent, to produce undesirable specifications. This is illustrated by an example from section 2.7.2 of [31] and shown in Figure 3 through an object diagram. An axiom stating that locks organized in a binary tree must be acquired prior to acquiring any of its children is inconsistent in the presence of tree mutation. This happens because the axiom is evaluated at the beginning of the method, i.e. the permissible locking orders are computed then. If the tree is mutated by changing its configuration, even if following the locking axioms, the resulting tree could have nodes that are in reverse position compared to the original tree, i.e. a parent could become a child of its original child. If a second modification were to happen inside the same method an attempt to follow the locking policy (acquiring the lock on the parent first and then on the child) would cause a violation to be reported. In this example, if the tree in Figure 3 moves from its initial state to its mutated state and back inside the same method while respecting the locking policy, the axiom in question would falsely report a violation upon acquisition of \( v \) after \( w \) since, originally, \( v \) was the parent and \( w \) the child. This happens because the axiom binds any free variables at the beginning of the method only and that is what dictates all acceptable orderings. During the reversion process, although \( w \) is then the parent of \( v \), the binding \( v < w \) is still in effect, thus causing a violation to be reported.

![Figure 3: Object diagram showing the two states of the tree. The mutated state depicts a rotation of the nodes from the initial state.](image-url)
The three issues above call for a better solution, which we present in section 4.4. The first issue precludes treating locking policies as part of a contract since it fails to provide a guarantee on the state subsequent to a method execution. The second issue affects the feasibility of implementing such checking during runtime in an efficient manner since such predicates would have to be evaluated at every step of the method execution due to their subject not being restricted to lock acquisition order. The third issue affects the correctness of the results.

### 3.6 Thread Safety

The core idea behind thread safety is one of non-interference [7]. Thread safety can be achieved in a variety of ways. They 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 issues. 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. Aliasing and ownership patterns that exist among objects also determine if accesses to a certain object are thread-safe or not. For instance, an object that is created inside a method and then returned to its client can be safely accessed if one knows that the creator class did not keep a reference to it (i.e. it is not part of the provider object representation). Serializability criteria for a region of code can be used to establish if the execution of the statements of a method in the presence of concurrency (i.e. possibly interleaved with statements executed by other threads) is equivalent to a sequential execution. A method is said to be atomic if the statements in the method body are serializable. The atomicity of a method execution ensures that sequential reasoning can be applied to it, i.e. it can be considered as executing in a single atomic step.

JML provides several constructs to specify all the above aspects of thread safety⁷. The expression `\texttt{thread_local}` states that its argument is owned by the current thread, i.e. it can only be reached by this thread. The `\texttt{monitors_for}` clause, in the format

---

⁷ Although described in [7], the following constructs have not yet been implemented on the JML checker or on the JML compiler (RAC), as of release 5.4 of the JML tools: `\texttt{thread_safe, thread_local, lock_protected, independent, and the atomic modifier.}` The 5.4 release happened on February 8th, 2007.
monitors_for <ident> <- <store-ref-list>, specify that all non-null locks specified in store-ref-list must be held prior to accessing ident. A simplified version of this clause is the `monitored` field modifier (see Listing 5 for an example), which states that a lock on the object containing the field instance, i.e. `this`, must be held prior to accessing the field. A stronger construct is the `\lock_protected` predicate, which states that access to the object provided as argument is protected by a non-empty set of locks, although it does not specify which locks in particular. The `\thread_safe` predicate is defined as the disjunction between `\thread_local` and `\lock_protected`, i.e. an object is considered thread-safe if it is either local to a thread or access to it is protected by a lock.

JML provides a number of constructs to deal with locks explicitly [7]. The `\lockset` expression refers to the set of locks held by the current thread. The `locks` clause in a contract specifies which locks a method acquires and releases during its execution. A method is not allowed to acquire more locks than what is present in the locks clause for the applicable specification case. The contract for `Pipe.get()` can be extended to include this clause as shown in Listing 8. Thus, it needs to be present in all cases for which it is relevant. The `\lockset` expression is of JMLObjectSet type. This is one of many modelling types provided by JML [8]. It is a set to which membership is based on object identity. Like other modelling types, its objects are immutable. One can obtain subsets, iterate over its contents, check membership, etc. but it cannot be modified.

There are situations in which it is useful to leave to the client the task of acquiring the necessary locks prior to executing a particular operation. This usually happens in cases the provider object is aware of concurrency issues but a per-operation concurrency control is not meaningful or practical. In such cases, concurrency control is delegated to another object (or set of objects) and it is the client’s responsibility to acquire the locks from such object prior to invoking the desired operation or set of operations. It is then easier to specify such behaviour directly in terms of locking requirements than through

---

8 Although described in the referred article, the `locks` clause has not yet been implemented on the JML checker or on the JML compiler (RAC), as of release 5.4 of the JML tools.
thread-safety predicates. A typical example of this case happens in transaction processing. The client must obtain a transaction lock prior to executing a set of operations on a set of objects and release it in the end. The operations performed while holding such lock are executed atomically. It does not make sense for the operations themselves to acquire the transaction lock but they “know” they must be executed in the context of a transaction. Another example in which it is easier to specify locking requirements than thread-safety predicates is when one uses well-known coarse grained locks to protect a set of objects. It is then very clear that a client must acquire a particular lock before executing a particular operation.

Aliasing control is done via the Universe type system embedded in JML. By using the rep modifier, one can specify that a particular object on a field declaration is part of the representation of another. This means that there should be no way to reach such object from outside the enclosing class. The readonly modifier prevents that the object pointed to by a reference be modified through it.

The atomic modifier can be applied to methods and classes. It flags that a method or a class is atomic. An atomic class is one in which all its methods are atomic. The atomic modifier is not inherited by subclasses.

These constructs were devised having static checking, theorem proving or even model-checking as the target verification techniques. There are several challenges in adapting them to runtime assertion checking. These can be summarized as the impossibility (or at least infeasibility) of stopping all threads of execution to evaluate reachability predicates. For instance, to decide if an object satisfies the predicate \texttt{thread\_local}, one would have to compute all references to such object and make sure they are all rooted on the same (the current) thread. Evaluating the \texttt{lock\_protected} predicate presents a similar challenge. With the monitors\_for clause, things are simpler for the common cases, but as complex in general. If modular reasoning is possible, i.e. the visibility of the monitored field is restricted (private), it is just a matter of inserting code (instrumenting) to check if the required locks have been acquired by the current thread, as long as there is

---

9 Method atomicity can also be checked using dynamic techniques. See [7] and [16] for more details.
no representation exposure. In such case, or if the field is accessible, external accesses to the referred object cannot be controlled by the container class, so such predicates would have to be constantly monitored or simply not at all.

Atomicity is in a different category. The techniques described in [7], [16] and [17] use pure reasoning, purely dynamic techniques or both, respectively. None of them can be easily adapted to a RAC scenario in which a predicate can be simply evaluated at the beginning or end of a method execution and a decision reached regarding its atomicity. The atomicity modifier is more of a language statement than a predicate to be checked. It simply informs that sequential reasoning can be used for the method in question. Since the focus of this research is on RAC, no atomicity related construct will be further considered.

The challenges presented above may limit the implementation of such constructs in a RAC compiler but the fundamental problems are the evaluation point and their use in the presence of specification inheritance. The identification of these challenges and problems is one of our contributions.

Locking requirements are specified through the use of the \lockset construct and checking for membership of a particular object in this set. Thread-safety is specified via the \thread_safe predicate on an object. Such predicates are currently specified in the precondition of a method. 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 target method. In JML, this is done in the pre-state, before entering the method but after binding the parameters. It could’ve been done, however, as the first statement of the method. There is no difference between these approaches for sequential programs. For concurrent programs, however, there are. The first statement of a method might already be protected by a monitor lock (synchronized methods in Java). Let’s take the method get() of class Pipe (Listing 8). It specifies a precondition containing locking requirements and a functional requirement. This implies that not only interference has an influence on the determination of the locking requirements to evaluate pre- and postconditions, but also the fact that obtaining such locks (whatever the strategy) might
affect the evaluation of the corresponding predicate. For instance, suppose one decides to evaluate the precondition of the method \texttt{get()} in the method pre-state. In this case, it is subject to external interference because it does not hold a lock on \texttt{this}, which protects the access to the field \texttt{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 \texttt{this} but before executing the \texttt{while} statement, interference is not a problem but the term \texttt{!\lockset\ .has(this)} will evaluate \texttt{false}, which is not the desired behaviour.

```java
/*@ 
 public normal_behaviour 
 requires closed &\& !\lockset\ .has(this); 
 locks this; 
 ensures \result == null; 
 also 
 public normal_behaviour 
 requires !closed &\& !\lockset\ .has(this); 
 when !isEmpty(); 
 locks this; 
 ensures \result != null; 
 */
 public synchronized Message get() throws InterruptedException 
 { 
   while (buf.isEmpty() &\& !closed) 
     wait(); 
   
   //@ commit: 
   if (closed) 
     return null; 
   
   return (Message) buf.removeFirst(); 
 }
```

Listing 8: The extended contract for the \texttt{get()} method of the class Pipe.

Locking requirements specifying that a particular lock is not held by the current thread will likely not hold even if they were true in the pre-state. The \texttt{\thread\_safe} predicate presents the reverse problem. It is possible for an object (e.g. a method argument) to not be thread-safe in the pre-state but be thread-safe inside the method due to the protection created by the monitor lock. One should note that the validity of the effective precondition does not change if evaluated in the pre-state or inside the method. This is the first evidence that locking requirements and thread-safety specifications do not

---

10 The \texttt{\thread\_safe} predicate is defined as the disjunction of the \texttt{\thread\_local} and \texttt{\lock\_protected} predicates. Our discussion assumes the case in which \texttt{\thread\_local} is not satisfied since its evaluation is not affected by the use of safepoints.
belong in the precondition. The ability to move the evaluation point of pre- and postconditions is essential to our solution as described in section 4.1.

The second problem is related to specification inheritance and behavioural subtyping. A predicate involving locking requirements or thread-safety is not required to be satisfied in a sub-type as long as the inherited specifications are satisfied. This is due to the fact that the effective precondition of a method is the disjunction of all inherited preconditions with the precondition of the target type. If that was not the case behavioural subtyping would be violated. The reverse case can also happen. A sub-type might weaken the precondition in a way that inherited locking requirements and thread safety specifications are not required to hold. This is the main point for not having these properties in the precondition. An analogous argumentation can be made for postconditions.

The main point is that preconditions, postconditions and invariants deal with the functional facet of objects. Thread safety properties cannot be easily combined with functional properties since the reasoning mechanisms utilized by the functional facet of objects produce counter-intuitive results. Our contributions are identifying this issue and presenting a solution (see section 4.2) by decoupling the functional and thread safety facets of contracts.

4 SPECIFYING CONTRACTS IN THE PRESENCE OF CONCURRENCY

The previous section explored the various problems encountered with using contracts to specify typical behavioural patterns of a concurrent object-oriented program.

This section presents a solution to the issue of interference and some of the instrumentation issues that are related to the semantics of contracts. It does not cover the problem of generating RAC code for each construct. Atomicity is never assumed to hold in any of this discussion. This complicates the argumentation but allows it to be more generic.
4.1 Safepoints

We start with a presentation of what a safepoint is followed by a discussion on how it can be used to solve some of the issues presented in section 3.3.

A safepoint is a point inside the method body at which it is safe to evaluate precondition or postcondition predicates together with object and class invariants. A method can have multiple safepoints. A precondition safepoint marks a point in which it is safe to compute the formulas of all preconditions and invariants, and the pre-state formulas of the selected postcondition. A postcondition safepoint marks a point in which it is safe to compute the formula of the postcondition selected at the time of the precondition safepoint and the invariants. Notice that this is the only postcondition for which thread safety is required to hold. No guarantees are made with respect to other postcondition formulas present in the method specification. A method execution path (from the first executable statement to the point in which it returns) can have only one precondition safepoint and only one postcondition safepoint. If no precondition safepoint is explicitly specified for a method, it defaults to the method pre-state. If a postcondition safepoint is not explicitly specified for an execution path, it defaults to the method post-state.

We propose the addition of the requires_safepoint and ensures_safepoint labels to JML to demarcate precondition and postcondition safepoints, respectively. Listing 9 exemplifies the use of such constructs and how they avoid the issue of internal interference. At the precondition safepoint, all the objects referenced by both requires clauses and the contents of the \old statements in the ensures clauses are properly protected by locks. At the postcondition safepoint, the field head, present in the ensures clause of the second specification case, is properly protected by a lock. Since result refers to the local variable x, it is also thread-safe at the postcondition safepoint. Finally, the object invariant can be safely evaluated both in the pre- and postcondition safepoints since it refers to the field head, which is properly locked in both places. Although not present in the specification for clarity purposes, access to field last is protected by (a lock on) head.
Listing 9: JML specification for the method extract of the LinkedQueue class using safepoints to avoid internal interference.

Having the precondition of a method, which is a predicate that specifies what needs to be satisfied prior to entering a method, evaluated inside such method is a counter-intuitive notion. A precondition, as initially presented by Meyer, can be thought as being 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 one knows that nothing significant to the evaluation of the precondition formula happens between these two states. The same idea applies to the precondition safepoint: nothing significant to the evaluation of the precondition predicate happens between the method’s pre-state and the safepoint. That is, 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), thus allowing safe evaluation of expressions referring to shared data and ensuring proper
correlation between the result of the precondition evaluation and the expected postcondition.

Analogously to preconditions, post conditions can be thought of being evaluated just before the method exits or just after it returns or throws an exception (i.e. in the post-state). Postcondition safepoints are another point to evaluate the postcondition predicate because nothing significant happens between such point and the methods’ post-state. One must note that the postcondition safepoint is obviously located before the method returns. This brings up the issue of how to evaluate the postcondition expression since it refers to $\text{result}$, the method’s return value. The postcondition safepoint must be the label of the $\text{return}$ or $\text{throw}$ statement. The expressions in such statements must be side-effect free. This can be easily checking at compilation time by requiring side-effect free only operations and purity from invoked methods and constructors. In case the method does not return a value, the $\text{ensures_safepoint}$ label can be placed at the end of a block or just before the method returns.

Two issues were purposefully left out of the above arguments: formula involving method parameters and what statements are considered significant to the evaluation of the preconditions, postconditions and invariants. The former relates to thread-safety requirements, which are covered in the examples to follow. The latter is deferred to the section on formalization.

The example above shows how safepoints can be used to solve the issue of interference. Listing 10 shows how a precondition safepoint can be used to prevent external interference from causing problems in the presence of blocking behaviour (see 3.4). No extra safepoint is needed for the $\text{when}$ clause, since it is evaluated at the commit point, which is already located in a thread-safe place. The issue this safepoint addresses is the case in which external interference causes the wrong $\text{when}$ clause to be evaluated. The safepoint ensures that the values of $\text{closed}$ observed by the formula and the method body are the same by moving the precondition evaluation point to inside the method body where it is not possible to change the value of $\text{closed}$ between the precondition evaluation and its observation by the current thread. This example also shows the case in
which no explicit postcondition safepoint is necessary. Since the postcondition formulas
do not reference shared data they can be safely evaluated in the method’s post-state.

```java
/*@  
   public normal_behaviour  
   requires closed;  
   ensures \result == null;  
   also  
   public normal_behaviour  
   requires !closed;  
   when !isEmpty();  
   ensures \result != null;
*/

public synchronized Message get() throws InterruptedException {
    //@ requires_safepoint:  
    while (buf.isEmpty() && !closed)
        wait();

    //@ commit:  
    if (closed)
        return null;

    return (Message) buf.removeFirst();
}
```

Listing 10: Using safepoints to prevent external interference in the presence of
blocking behaviour. Excerpt of Listing 5 showing only the modified method.

Safepoints can also be used to prevent interference in the presence of specification
inheritance. Listing 11 shows how safepoints can be used to prevent external interference
for method receive() of class PipedChannel (see Listing 4). The specification for
this method is inherited from the interface it implements (see Listing 3). A safepoint
defines the place at which to evaluate the resulting specification (i.e. pre- or
postcondition and invariants, including history constraints), which is the desugared
specification taking into account all inherited specifications.
public Message receive() throws NotConnectedException {
    if (closed) {
        //@ requires_safepoint:
        throw new NotConnectedException();
    }
    if (in.isEmpty()) {
        synchronized (this) {
            //@ requires_safepoint:
            if (remoteClosed) {
                closed = true;
                throw new NotConnectedException();
            }
        }
        return null;
    }
    //@ requires_safepoint:
    return in.take();
}

Listing 11: Method receive() of class PipedChannel equipped with safepoints.

Both specification cases reference model fields connected and remoteClosed. The values for these fields are obtained through the represents clauses defined in class PipedChannel. The model field connected is realized through fields closed and remoteClosed of class PipedChannel. The evaluation of connected needs to occur at a place where the concrete fields are protected from interference or after their values have already been obtained. An example of the former case is the second safepoint in Listing 11. 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 latter case is the first safepoint in the same method. It is located after closed evaluates to true. By looking at the source code of the whole class (not shown here) one can see that once closed is set, it is never reset. Thus the safepoint is in a safe place to evaluate the precondition in case connected is false. Locking is not required in this case because Java guarantees that accesses and
assignments to variables of type int and boolean are atomic. Otherwise, some locking would be required to guarantee atomic access to these fields.

There are two details about this example that need special attention. A closer look at the specification of receive() in Listing 3 shows a problem that is not addressed by the use of safepoints alone: the postcondition \( \text{result} \neq \text{null} \implies n\text{Pending} = \text{old}(n\text{Pending}) - 1 \) cannot be always satisfied in case the left side of the implication is true. If another Message is put into the Pipe referenced by in after it returns but before the postcondition is evaluated in the method’s post-state, the size() of in (which realizes nPending) will not be one less of what it was at the precondition safepoint (the third one, in this case). Even if one were to add a postcondition safepoint right before the return statement, it would not be enough. This is due to the fact that the concurrency control happens inside the Pipe.get() method. Once it returns, thus releasing the lock on the Pipe, another thread is free to execute Pipe.put(). If one wanted to enforce the postcondition in question, one would have to acquire a monitor lock on in and include the pre- and postcondition safepoints together with the call to Pipe.get() in this synchronized block.

The other detail is regarding the history constraint. A history constraint is a type of class invariant that specifies predicates regarding state transitions. In this example, it specifies that once connected becomes false it cannot become true anymore. The value of \( \text{old}(\text{connected}) \) is computed together with the precondition safepoint. The value of connected is computed together with the postcondition safepoint, which in this example is in the method’s post-state. Although not strictly thread-safe, it is safe to evaluate connected in the post-state due to the atomicity guarantees offered by the JVM. More sophisticated constraints might have more stringent locking requirements, which would affect the use of safepoints. It should be nevertheless clear that evaluating a constraint at a postcondition safepoint would entail the same result as in the post-state in an interference-free environment since nothing significant can happen between these two places.
Safepoints can also be applied to contracts exhibiting the combined behaviours of blocking behaviour and specification inheritance. Listing 12 shows such an example from section 3.4. By holding a lock on Pipe $in$ and having the precondition safepoint located inside this monitor, one can prevent the issues of internal and external interference. The Pipe is locked while its number of elements is checked, the model field $n_{Pending}$ is evaluated, and the preconditions are evaluated determining the postcondition to be evaluated at the postcondition safepoint and the when clause to be evaluated at the commit point. Notice that the super-class (i.e. PipedChannel) implementation of $\text{receive()}$ is executed while holding a lock on $in$. This prevents the issues mentioned above regarding the satisfiability of the postcondition involving the number of messages in the channel for the nested call.

The use of safepoints does not solve the problem of breaking behavioural subtyping created by introducing blocking behaviour (see section 3.4 for more details). Even though interference is avoided, the precondition safepoint cannot prevent the Pipe from receiving messages while $\text{receive()}$ is blocked. This can only be solved by relaxing the specification of $\text{Channel.receive()}$ to offer weaker guarantees.
Listing 12: Method `receive()` of class `BlockingChannel2` equipped with safepoints.

4.1.1 Discussion

Up to this point we only discussed cases in which contracts refer to the internal state of an object and the effective precondition is `true`. The problems safepoints solve are correlating preconditions to postconditions and guaranteeing thread-safe access to objects present in such formulas. In other words, they exist to demarcate a point in the code in which the value of the variables the pre- or postcondition predicate observes is the same as the one the method execution does. 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 defects on the final (uninstrumented) program. Instead, the idea is to find the right place in the code to evaluate the contract following, to the extent possible, the informal reasoning 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()` (see Listing 9) 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. 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. Placing the safepoint incorrectly will cause similar problems as not dealing with a valid case or generating an unexpected result for a valid input.

Of course, the placement of safepoints needs to follow the rules defined in the beginning of this section to guarantee the expected semantics of a contract, i.e. that preconditions and postconditions are evaluated only once for a method execution. These constraints are checked during compilation time by doing a simple flow control analysis to guarantee that only one precondition and one postcondition safepoint per execution path can be executed and that only unobservable statements exist between the beginning of a method and a precondition safepoint and between a postcondition safepoint and the end of a method. The technique to perform such check is very similar to what is used to guarantee that a constant variable declared at the beginning of a method is only initialized once. Of course, such technique discards valid cases since it is undecidable to determine if a safepoint is going to be executed more than once, in general.

The use of safepoints allows for the elimination of non-determinism in the specification of a concurrent method. Although a valid specification technique, the use of non-determinism on a RAC scenario defeats the purpose since it would reduce the detail of the specification. The more detail present on a specification, the higher the likelihood of detecting a fault. Comparing with existing techniques [43][44], in which all variables referenced in specifications are required to be thread-safe, all the preconditions on the examples in this section would have to be reduced to `true`, i.e. no requirements, since
all of them refer to non-thread-safe variables. We see value in being able to specify such properties.

4.2 Specification of Thread-Safe Behaviour

The specification of thread-safety properties can be divided in two types: locking requirements and thread-safety predicates. The former type deals with the specification of locks that must be held by the current thread to execute a particular method. The latter type deals with thread-safety of the objects used during the execution of a given method. As discussed in section 3.6, JML already provides constructs that allow the specification of these properties but these constructs present shortcomings we consider to be of fundamental importance in allowing their use in the presence of specification inheritance. In this section we present our solution to them emphasizing their use for RAC in conjunction with safepoints. Our solution considers all the challenges mentioned in section 3.6.

The fundamental issues with the current approach to the specification of thread-safe behaviour are the predicate evaluation point and their meaning in the presence of specification inheritance. The root cause of these issues is the mixing of thread safe behaviour specification with functional specification.

Functional specifications deal with state transformations. They specify properties that must hold on the states preceding and following a method execution. Thread safety specifications deal with the properties that must hold to ensure that such state transformation occurs as specified in a concurrent environment. A program can, then, be seen as the combination of two facets: the functional and the concurrent. The functional facet is the one that deals with retrieving, processing and outputting data. The concurrent facet is the one that deals with the mechanisms to guarantee that access to such data by multiple threads is controlled. The functional facet specifies properties that depend on the client and the provider. The concurrent facet specifies properties that depend on the environment. The environment is everything else that happens outside of the current thread’s control. The current thread affects the environment through the acquisition and release of locks. These concepts are so independent that one usually states informal
requirements about thread safety independently of functional properties. Method specifications should reflect this independence.

The concurrent facet of a method specifies its thread-safety requirements and assurances. These predicates can take the form of the \texttt{thread\_safe} predicate as well as the form of locking requirements and assurances. Both styles serve the same purpose: to specify the conditions for the thread-safe execution of the target method. Depending on the situation, it is easier to specify directly that a particular lock must be acquired before a method execution than to state that all objects this method refers to are \texttt{thread\_safe}. There are situations in which locking properties are functional properties. For instance, in a lock server class, the act of acquiring a lock and returning it on a method that is supposed to do just that should be treated as part of the functional facet since it is used to fulfil the method’s functionality. Specifications should reflect this dual purpose of locking properties.

We propose the addition of the \texttt{concurrent\_behaviour} specification to a method specification. The syntax of this specification is given by the productions in Figure 4 following the conventions in [9]. The production \texttt{specification} is extended with the addition of the optional production \texttt{concurrent-spec}. The other productions of \texttt{specification} are unchanged and therefore not reproduced in here. Refer to section A.6 of [9] for the complete production.

\begin{verbatim}
specification ::= spec-case-seq [ redundant-spec ] [ concurrent-spec ] | redundant-spec
concurrent-spec ::= concurrent-spec-keyword concurrent-spec-body
concurrent-spec-body ::= concurrent-spec-body-clause
                        [ concurrent-spec-body-clause ] …
concurrent-spec-keyword ::= concurrent\_behaviour | concurrent\_behavior
\end{verbatim}

\textbf{Figure 4: The modified \texttt{specification} production including the new \texttt{concurrent-spec} production}

We propose the addition of the following clauses to the method specification:
• `requires_locked`, `requires_unlocked`, `ensures_locked`, `ensures_unlocked`: specifies that the set of lock objects provided are all held or not held by the current thread, respectively, in the method pre-state and post-state, respectively. Null references are ignored.

• `requires_thread_safe`: specifies that all the objects provided satisfy the \texttt{thread\_safe} predicate in the method pre-state. Null references are ignored.

• `ensures_thread_safe`: specifies that all the objects provided satisfy the \texttt{thread\_safe} predicate in the method post-state. Null references are ignored.

We also make the locks clause part of the concurrent facet instead of the functional facet as originally proposed in [7]. This clause is intrinsically part of the concurrency control aspect of programs since it is related to independence\textsuperscript{11}. In this context, it is meaningless to allow for different specification cases to specify different locks clauses. This would mean that the client would have to either acquire the locks in the union of the set of locks specified in all specification cases or to selectively acquire the locks based on the specification case it is expecting to be satisfied. The former case trivially reduces to specifying all the locks in the concurrent facet. The latter inductively reduces to the same situation for two adjacent methods to be treated as a single independent portion of code require that locks clauses from both to be satisfied. In such case, the client would have to acquire the locks from the union of the individual sets of locks specified on both methods’ specifications.

The syntax of these clauses is given by the productions in Figure 5 following the conventions in [9]. The production `spec-body-clause`\textsuperscript{12} is extended by having `locking-body-clause` as a new alternate production. This allows using the locking clauses in regular (functional) specification cases. We decided for using the keyword `locks_releases` instead of the proposed `locks` because requiring it to be a reserved

\textsuperscript{11} See [7][14] for more details regarding independence. The execution of an independent statement or region is performed in an atomic fashion. Two adjacent independent regions can be thought of as a single independent region.

\textsuperscript{12} The production `spec-body-clause` is defined in section A.6 of [9].
keyword would have made most JML programs impractical since this would forbid referring to such identifier inside a JML annotation. One such instance would be the use of a model import directive related to the `java.util.concurrent.locks` package.

```
concurrent-spec-body-clause ::= requires-thread-safe-clause |
                             ensures-thread-safe-clause |
                             locking-body-clause |
                             locks-clause ;
locking-body-clause ::= requires-locked-clause | ensures-locked-clause |
                      requires-unlocked-clause | ensures-unlocked-clause ;
requires-locked-clause ::= requires-locked-keyword spec-expression-list ;
requires-locked-keyword ::= requires_locked
ensures-locked-clause ::= ensures-locked-keyword spec-expression-list ;
ensures-locked-keyword ::= ensures_locked
requires-unlocked-clause ::= requires-unlocked-keyword spec-expression-list ;
requires-unlocked-keyword ::= requires_unlocked
ensures-unlocked-clause ::= ensures-unlocked-keyword spec-expression-list ;
ensures-unlocked-keyword ::= ensures_unlocked
requires-thread-safe-clause ::= requires-thread-safe-keyword spec-expression-list ;
requires-thread-safe-keyword ::= requires_thread_safe
ensures-thread-safe-clause ::= ensures-thread-safe-keyword spec-expression-list ;
ensures-thread-safe-keyword ::= ensures_thread_safe
locks-clause ::= locks-keyword spec-expression-list ;
locks-keyword ::= locks_releases
```

**Figure 5:** The productions specifying the newly introduced clauses.

Every expression in the `spec-expression-list` productions above must evaluate to an object reference. The semantics of a method specification that includes a `concurrent_behaviour` specification is the following:

---

13 The production `spec-expression-list` is defined in section A.8 of [9].
• requires_thread_safe, requires_locked and requires_unlocked must be satisfied in the method’s pre-state. In case of inherited specifications, the effective specifications for these clauses must be satisfied. The inheritance semantics is discussed below. With respect to RAC, these clauses must be evaluated and satisfied prior to any requires clause (functional precondition) is evaluated.

• ensures_thread_safe, ensures_locked and ensures_unlocked must be satisfied in the method’s post-state. In case of inherited specifications, the effective specifications for these clauses must be satisfied. With respect to RAC, these clauses must be satisfied for the implementation to be considered correct but they do not relate to the ensures clauses (functional postconditions), which can be evaluated prior to the post-state (i.e. a postcondition safepoint).

• locks_releases must be satisfied in the method’s post-state. In case of inherited specifications, the effective specification of this clause must be satisfied.

If locking clauses are used in the functional facet of the specification, they are treated as regular predicates and subject to the standard specification inheritance and desugaring rules. The only difference is that they are not evaluated with their respective safepoints. requires_locked and requires_unlocked are always evaluated in the pre-state, and ensures_locked and ensures_unlocked are always evaluated in the post-state of a method execution.

All concurrent specification clauses default to not_specified. The semantics of specification inheritance on the concurrent facet is exactly as for invariants. The effective specification of a particular clause 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.

Thread-safety specifications, like invariants, should only be strengthened by sub-types. At first, one might think that such properties should not be inherited at all since concurrency control is very particular to a type. Usually one would not see these
properties as exposed functionality but as implementation details. However, it should be possible for a class to specify these properties for associated objects designed for extension. A typical example is a class in a messaging framework. The framework could define a Message interface and state locking and thread-safety requirements for implementers so that they can be handled without risk of deadlocks or race conditions. Another point for the proposed semantics is that there are cases in which 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 from the concrete message since it knows it will only process RPC messages.

The idea is that concurrent specification clauses specify environment invariants. Environment invariants specify properties that the environment (i.e. threads) must respect before and after a method execution. The only difference is that (functional) invariants are concerned about the object’s state as seen by the current thread and environment invariants are concerned about the state of the environment as seen by the target object. These properties deal with the state of locks, which is the way threads interact. The target object observes the environment through such locks and determines if the environment is safe for the current thread to execute its method.

The declaration of method sendAndWait() of class BlockingChannel in Listing 15 demonstrates the use of thread-safety specification clauses. The requires_thread_safe clause specifies that the object aRequest refers to must be thread-safe in the method pre-state. This clause must hold on the method pre-state. Similarly, ensures_thread_safe specifies that the object returned by the method execution must be thread-safe on the method’s post-state. The use of locking clauses in the concurrent facet is analogous.
The example on Listing 14 shows the use of locking clauses on the functional facet and the use of the locks_releases clause according to our proposal. This is a modification of the example on Listing 8. This is just a syntactic example. The requires_unlocked clauses in both specification cases are evaluated in the method pre-state. They can be considered as being conjoined to the precondition of its associated specification case. The locks_releases clause on the concurrent facet specifies that the method must acquire and release a lock on this.
/*@
   public normal_behaviour
   requires closed;
   requires_unlocked this;
   ensures \result == null;
   also
   public normal_behaviour
   requires !closed;
   requires_unlocked this;
   when !isEmpty();
   ensures \result != null;
   concurrent_behaviour
   locks this;
*/

public synchronized Message get() throws InterruptedException {
    while (buf.isEmpty() && !closed)
        wait();

    //@ commit:
    if (closed)
        return null;

    return (Message) buf.removeFirst();
}

Listing 14: The contract for the get() method of the class Pipe using the new locking constructs.

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

4.3 Thread Safety + Safepoints = No Interference

This section presents how one can combine the concepts of thread-safety with safepoints to solve the problem of interference. We describe this informally and then present a semiformal description of the semantics of a concurrent contract.

Safepoints alone cannot solve the issue of thread-safety when the effective precondition is not simply true, i.e. the client is required to establish a certain state prior to executing
a method on the provider. Such state can take the form of predicates on method arguments or predicates involving the internal state of the provider.

Listing 15 presents a simple contract in which the effective precondition involves predicates on method parameters. The effective precondition of method \texttt{m3()} is \texttt{o2.p1().} Since it is the responsibility of the client to establish such predicate, it is also reasonable to expect that it provides conditions for the provider to safely observe 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 \texttt{requires_thread_safe} clause on all objects participating in the effective precondition. In this example, \texttt{o2} is required to be thread-safe. Once such objects are thread-safe, the predicates on them can be checked at the safepoints since they will not change between the method pre-state and the precondition safepoint.

```java
/*@ normal_behaviour
  requires o.p() && o2.p1();
  ensures Q;
also normal_behaviour
  requires !o.p() && o2.p1();
  ensures S;
concurrent_behaviour
  requires_thread_safe o2;
@*/

public void m3(T o, T2 o2)
```

Listing 15: Example of contract involving thread-safety requirements of method parameters. Q and S represent valid JML predicates.

The requirements are similar with respect to postconditions. If such predicates involve any state that must be established by the provider and observed by the client, the associated objects must be flagged as thread-safe. If such objects are method parameters, then they should be covered by the \texttt{ensures_thread_safe} clause. If the return value is one of these objects, \texttt{\result} must be included in the \texttt{ensures_thread_safe} clause. A typical example of this case is a method that returns a collection and the postcondition specifies a membership predicate. Such collection must be thread-safe for such predicate to make sense.
The only meaningful case in which the internal state of the provider is present in the effective precondition in a concurrent environment is related to method invocation ordering requirements. This case is solved by using the locks clause on this or any specification accessible field designated as a lock protecting the explicit or implicit state machine implemented by such class in combination with safepoints to demarcate the safe place to check for the ordering predicate. Listing 16 exemplifies this situation. Method ev1() can only be executed if the object is in state S1 and it changes the state to S2. The locks_releases clause tells the client that holding a lock on this guarantees the independent execution of the method. There are two basic forms of ensuring the correct use of such object. The client can be designed in such a way that it can never attempt an illegal transition (e.g. through a rendezvous mechanism between different threads) or competing threads can acquire a lock on the StateMachine object and call the event methods ev1() and ev2() while in this monitor. The locks clause guarantees independence to the client.
```java
public class StateMachine {

    public static final int S0 = 0, S1 = 1, S2 = 2;

    private /*@ spec_public monitored @*/ int state = S0;

    /*@
        normal behavioural
        requires state == S0;
        ensures state == S1;
        locks this;
    */

    public synchronized void init() { ... }

    /*@
        normal behavioural
        requires state == S1;
        ensures state == S2;
        locks this;
    */

    public synchronized void ev1() {
        //@ requires_safepoint:
        // do stuff

        state = S2;
        //@ ensures_safepoint:
    }

    /*@
        normal behavioural
        requires state == S2;
        ensures state == S1;
        locks this;
    */

    public synchronized void ev2() { ... }
}
```

Listing 16: Sample state machine designed for a concurrent environment. State changes are protected by synchronized methods.

The combination of thread-safety requirements for state that is established by the client to be observed by the provider and state that is established by the provider and observed by the client with safepoints, which demarcate a point for the thread-safe evaluation of expressions related to the internal state of the provider and objects passed as method parameters, guarantees freedom from interference. This has been presented informally through several examples in this section. We now present a semi-formal description of
the semantics of concurrent contracts so one can understand their meaning and limitations.

4.3.1 Informal Semantics of Concurrent Contracts

**Definition 1 (Unobservable Statement):** The execution of a statement inside a method body is said to be unobservable if its effects do not change any state that can be observed by any user thread or it changes state local to the method body to which such statement belongs and it does not cause the method to terminate unless it does so by throwing a `java.lang.Error` exception. A user thread is the main thread or any thread created by the execution of a statement of a user thread.

The idea behind an unobservable statement is that it does not produce any side-effect and does not interrupt the flow of control. The method terminating by throwing a `java.lang.Error` is not considered an observable event because this exception signals an internal error of the Java Virtual Machine and the method is, thus, released from any obligations (see section 9.6.2 of [9]). This is considered an unrecoverable condition [29]. This definition is restricted to user threads because system threads like the garbage collection threads are not under program control and can observe any state at any point in time.

**Lemma 1:** The following statements are unobservable: assignment to local variables, evaluation of side-effect free expressions, if ... else, loops, try block, switch statement, break and continue. For the compound statements mentioned above, they are unobservable only if their substatements are unobservable.

*Proof:* It follows immediately from definition 1.

**Lemma 2:** The following statements are unobservable: pure method or constructor call, object allocation.

*Proof:* Pure methods are, by definitions, side-effect free. A pure method is not allowed to perform any assignments to any fields or to execute a non-pure method or constructor. A
pure constructor can only assign to fields of the newly allocated object and perform pure method or constructor calls. Therefore, a pure method cannot perform any changes on the state of any object and thus, its execution is unobservable. A pure constructor is only establishing the state of an object that has been newly allocated which can only be observed by the local method body and is, thus, unobservable. Object allocation can only be observed by the internal JVM threads responsible for garbage collection and is, thus, unobservable.

**Lemma 3**: The following JML constructs are unobservable: annotation statements with the exception of the set statement (for ghost non-local variables), the assert statement, the assume statement, loop variant and invariant clauses.

**Proof**: By definition, the arguments of these constructs must be side-effect free expressions, including pure method calls or constructors. Therefore, the evaluation of their arguments is unobservable. The evaluation (in a RAC context) of these statements produces either:

1. No result, in the case of loop variants, loop invariants, assert, assume and annotations other than the set statement if the predicate is satisfied.

2. An assertion error being thrown in the case of loop variants, loop invariants, assert, assume, and annotations other than the set statement if the predicate is not satisfied.

3. An assignment to a local ghost variable.

For cases 1 and 3, it follows immediately from Definition 1. For case 2, assertion errors are subclasses of `java.lang.Error` [11] and it follows from Definition 1.

**Lemma 4**: The synchronized statement is unobservable in a deadlock free program.

**Proof**: Monitor locks are acquired and released by the use of synchronized statements. Monitor locks are reentrant, thus once a thread allocates a certain lock, it is guaranteed
that it will succeed in acquiring it again (i.e. entering into a synchronized block). A thread succeeds in entering a synchronized block if the lock object has not been acquired by another thread. In a deadlock free program, every thread is guaranteed to progress, by definition. Therefore, every thread that enters a synchronized block is guaranteed to leave it. Thus, every thread that acquires a lock is guaranteed to release it. Thus, a user thread can enter and exit any number of synchronized blocks. Therefore, a user thread cannot observe any difference from executing a synchronized statement in which a lock is available and one in which it is not. Thus, it is unobservable.

**Definition 2 (Safepoints):** A precondition safepoint is a statement S identified by the requires_safepoint label in which the following holds:

1. A precondition safepoint is **definitely not executed before** S

2. The preceding statement is marked as unobservable

A postcondition safepoint is a statement S identified by the ensures_safepoint label in which the following holds:

1. A precondition safepoint is **definitely executed before** S

2. S is either a return statement, a throw statement or, for a method with return type void, the null statement at the end of a block after which the method is guaranteed to return normally.

3. If S is a return or a throw statement, S is composed only of unobservable expressions.

The notion of definite execution derives from the rules of definite assignment of Java (see chapter 16 of [48]). Every method can be thought of as declaring a local final variable named preSafe as its first statement. If the method does not contain any requires_safepoint label in its body then preSafe is assigned in the statement immediately following its declaration. Otherwise, preSafe is assigned a value at every requires_safepoint label for a statement S. If such
assignment is legal according to Java’s assignment rules, then a precondition
safepoint is definitely not executed before S.

\textit{preSafe} is read at every statement S identified by the label
\texttt{ensures\_safepoint}, at a return statement, at a throw statement and at the
end of the method if it is of type \texttt{void}. If such accesses are legal according to
Java’s access rules, i.e. which only definitely assigned variables can be read then
a precondition safepoint is definitely executed before S.

A statement S is marked unobservable if:

1. It is an unobservable statement and its preceding statement is marked
   unobservable

2. It is an unobservable statement and it is the first statement in the method
   body

\textbf{Definition 3 (Concurrent Method):} A (well-formed) concurrent method is a
method that either does not require safepoints or the following holds:

For each precondition safepoint:

1. All the locks necessary to evaluate all the precondition and class invariant
   predicates in a thread-safe manner are held by the current thread.

2. All the locks necessary to evaluate all the \texttt{old} expressions present in all
   postcondition and history constraint predicates in a thread-safe manner are
   held by the current thread.

For each postcondition safepoint:

1. All the locks necessary to evaluate the postcondition associated with the
   specification case containing the satisfied precondition and class invariant
   predicates in a thread-safe manner are held by the current thread.
2. All the locks necessary to evaluate all the history constraint predicates in a thread-safe manner are held by the current thread.

A precondition safepoint is required if at least one precondition, class invariant or $\texttt{old}$ expression refers to an object for which access is not thread-safe or a consistent view of these objects is required for the correct evaluation of such predicates, and the required locks are only acquired inside the method body. A postcondition safepoint is required if the postcondition formula to be evaluated (as determined by the satisfied precondition) refers to an object for which access is not thread-safe or a consistent view of these objects is required for the correct evaluation of such predicates, and the required locks are only held inside the method body.

It should be noted that the definition above implies thread-safety of all method arguments for which locks are not acquired inside the method body. Although there is no requirement to specify such arguments as thread-safe in the concurrent facet of the method specification, it is recommended as a matter of style to make it so. It enables static checkers to verify their validity and directs the RAC to include assertion checking code for them\(^\text{14}\).

**Definition 4 (Effective precondition and Observable postcondition):** The effective precondition of a method is the predicate resulting from the minimization of the disjunction of all preconditions present in the method specification of the target class with all preconditions inherited from the method specification of its super-types.

The observable postcondition of a method is the predicate resulting from the minimization of the disjunction of all postconditions present in the method specification of the target class with all postconditions inherited from the method specification of its super-types.

\(^{14}\) The jmlc tool does not yet implement assertion checking code generation for this construct. It is unknown to the authors if there are any static checkers that make use of this predicate.
The definition of effective precondition adheres to the notion of behavioural subtyping and, therefore, requires no further clarifications. The observable postcondition, however, presents a counter-intuitive notion that requires discussion. The idea behind it is that the predicate a method establishes is what is common between all postconditions as if all preconditions were satisfied. This notion is instrumental to determine thread-safety requirements, as will be seen below.

In a well-formed and realizable (implementable) method specification, specification cases have either mutually exclusive preconditions or have postconditions that evaluate consistently (for behavioural subtyping); otherwise the effective postcondition would be unsatisfiable. For mutually exclusive preconditions, the disjunction of the associated postconditions removes the terms that do not matter, since the method can, depending on the precondition, establish either a term or its complement.

**Theorem 1 (Interference Freedom):** A concurrent method for which all objects referenced from the effective precondition and observable postcondition formula are thread-safe is interference-free.

*Proof:* For the case a concurrent method does not have any type and method specifications, it is trivially satisfied since no interference is possible, by definition. If safepoints are required, interference is prevented by the use of safepoints. This is so because precondition safepoints force the evaluation of preconditions to happen only at points in which access to all the referenced objects is protected by a lock. Objects referenced from specification cases that are not part of the effective precondition and the observable postcondition are part of the internal state of the target object or asynchronously established by the environment and, therefore, are protected by safepoints. Objects referenced by the effective precondition or the observable postcondition are protected on the call site due to the thread-safety requirements and, therefore, are not subject to interference at safepoints. If the client guarantees sequential execution, internal interference is not possible. External interference is prevented for the same reason as in the previous case reduced to a
case with no safepoints, i.e. preconditions and postconditions are evaluated in the method pre-state and post-state, respectively.

**Theorem 2 (Equivalence of Concurrent and Sequential Contracts):** A concurrent method for which all objects referenced from the effective precondition and observable postcondition formula are thread-safe is equivalent to a method with the same method and type specifications with safepoints and concurrent specifications removed, if executed in a sequential environment.

*Proof:* From definition 2, it is guaranteed that method preconditions are evaluated only once per method execution per construction. From the same reason applied to postconditions, it is guaranteed that the associated method postcondition is evaluated only once per method execution. Therefore, class invariants are evaluated twice per method execution: once with the precondition and once with the postcondition. History constraints are evaluated only at once together with the postcondition. Also, \( \text{old} \) expressions are evaluated only once, together with the precondition (I). Also from definition 2, the restriction that there are only unobservable statements from the beginning (end) of a method and a pre- (post-) condition safepoint guarantees that the method itself does not modify the outcome of the pre- (post-) condition and invariants if they were evaluated at the method’s pre- (post-) state (II). Theorem 1 guarantees that the evaluation of specification expression is consistent with the actual behaviour of the method (III). From I, II and III above, a concurrent contract for a concurrent method presents the same behaviour as the same contract without the concurrent features if applied to a method in a purely sequential environment.

To summarize this section, a concurrent contract can be applied to the same method in a sequential environment and they are guaranteed to evaluate the same. More loosely: a concurrent contract presents the same semantics of its sequential version. Notice, however, that it is not guaranteed that every sequential contract can be applied to a concurrent environment. Examples in previous sections (see section 3.4 and Listing 6)
presented situations in which some predicates could not be guaranteed to hold even in an interference-free environment without changing the method implementation.

4.4 Lock Acquisition Order Specification

This section presents our solution to the problem of specifying lock acquisition ordering policies for concurrent classes. In section 3.5 we pointed out several shortcomings of the current practice, namely the possibility of unexpected behaviour, the unnecessary flexibility causing unwanted complexity for the verification task, and the asymmetry of the construct that made it difficult to be considered as a class invariant.

We propose the addition of a type specification clause\textsuperscript{15} called \texttt{lock_order} to JML. The syntax of this clause is given by the productions in Figure 6, following the conventions in [9].

\begin{figure}
\begin{verbatim}
lock-order-clause ::= lock-order-keyword lock-order-expression-list ;
lock-order-expression-list ::= lock-order-expression [ , lock-order-expression ] …
lock-order-expression ::= spec-expression l-order-op spec-expression
l-order-op ::= < | <=
lock-order-keyword ::= lock_order
\end{verbatim}
\end{figure}

Figure 6: The productions specifying the lock order clause.

The semantics of the ordering operator is the following. Let \( l_1 \) and \( l_2 \) be instances of \texttt{java.lang.Object}. The expressions \( l_1 < l_2 \) and \( l_1 \leq l_2 \) evaluate, respectively:

1. \texttt{true} and \texttt{true}, if the current thread acquired lock \( l_1 \) before \( l_2 \) or if it acquired \( l_1 \) but not \( l_2 \).

2. \texttt{false} and \texttt{false}, if the current thread acquired lock \( l_2 \) before \( l_1 \) or if it acquired \( l_2 \) but not \( l_1 \).

\textsuperscript{15} A type specification clause is a production of \texttt{jml-declaration}. See section A.5 of [9]. All \texttt{modifiers} are applicable to this clause as well.
3. **false** and **true**, if the current thread did not acquire \text{l1} nor \text{l2}.

The type of the two *spec-expressions* must be either \text{java.lang.Object}, or \text{org.jmlspecs.models.JMLCollection}. In the latter case, the ordering operator is applied to each element of the collection and the *lock-order-expression* evaluates to **true** if, and only if, it evaluates to **true** for each such element. In case both spec-expressions are of type \text{org.jmlspecs.models.JMLCollection} the ordering operator is applied to every pair of elements according to the Cartesian product of the two sets. For instance, the *lock-order-expression* \(a < \{b, c\}\), where \(\{b, c\}\) represents a \text{org.jmlspecs.models.JMLCollection} with elements \(b\) and \(c\), is equivalent to two *lock-order-expressions* \(a < b\) and \(a < c\). Analogously, \(\{a, b\} < \{c, d\}\) is equivalent to: \(a < c, a < d, b < c, b < d\).

The semantics of the *lock-order-clause* is that each *lock-order-expression* must hold for every state it is in effect in the context of the current thread. A *lock-order-clause* is in effect for a given state if such state is in the activation record of a method belonging to the type declaring such a clause or one of its subtypes.

These concepts are better understood through an example. Let class \text{A} define lock order clause \text{La} and class \text{B}, \text{Lb}. Also, let objects \(a1\) and \(a2\) be instances of \text{A} and \(b1\) and \(b2\), of \text{B}, and interconnected as depicted in Figure 7. Let thread \(T\) call method \text{test()} to perform such initialization and then \(a1.m1()\) such that the execution on Figure 8 occurs (the methods in question can return at the several . . . . for different reasons that are not relevant to this example) as in Listing 17. Let \text{La1} denote \text{La} bound to object \(a1\), \text{La2} to \(a2\), \text{Lb1} to \(b1\), \text{Lb2} to \(b2\).
class A
{
    A a;
    B b;
    //@ lock_order La;
    void m1() {
        ....
        m2();
        synchronized(b) {
            ....
            a.m1();
        }
    }
    synchronized void m2() {
        ....
        b.m3();
        ....
    }
}

class B
{
    A a;
    //@ lock_order Lb;
    synchronized void m3() {
        ....
        a.m1();
        ....
    }
    public static void test() {
        A a1 = new A();
        A a2 = new A();
        B b1 = new B();
        B b2 = new B();
        a1.a = a2;
        a2.a = a1;
        a1.b = b1;
        a2.b = b2;
        b1.a = a2;
        b2.a = a1;
        a1.m1();
    }
}

Listing 17: Code fragments exemplifying the use of the lock-order-clause.

Figure 7: Object interconnection diagram to exemplify the semantics of the lock-
order-clause.
Figure 8: Sequence diagram depicting an execution instance of method a1.m1() for the running example. The notes attached to activations denote points in the execution referenced in the text and the lock order clauses in effect for such activation.

When thread \( T \) reaches point 1, the only lock order clause in effect is \( L_{a1} \). \( L_{a1} \) is still the only one in effect at point 2. When it gets to point 3, \( L_{a1} \) and \( L_{b1} \) are in effect, since \( T \) entered \( m_{3}() \) of \( b1 \) while still inside \( m_{2}() \) of \( a1 \). At points 4 and 5 we have \( L_{a1}, L_{b1}, L_{a2} \) in effect. At point 6, we have only \( L_{a1} \) since \( m_{3}() \) of \( b1 \) has already completed its execution.

The question, now, is what it means for a lock order clause to be in effect. It means that it is evaluated at every attempt of thread \( T \) in acquiring a lock.
Continuing on the example, let \( L_a \) be \( \text{this} < a \), \( \text{this} < b \), \( b \leq a \) and \( L_b \) be \( \text{this} < a \). At point 1, although \( L_{a1} \) is in effect, no locks have been acquired, so \( L_{a1} \) has not yet been evaluated. \( L_{a1} \) will be evaluated for the first time at point 2, just before the lock \( a_1 \) is acquired (as part of the synchronized method \( a_1.m2() \)). It evaluates to \text{true} since \text{this} is about to be acquired and the lockset for thread \( T \) does not contain \( a_2 \) or \( b_1 \), the objects referenced by members \( a \) and \( b \) of \( a_1 \). To reach point 3, \( T \) must acquire a lock on \( b_1 \), since \( m3() \) is synchronized. At that point, \( L_{a1} \) and \( L_{b1} \) are in effect, so both are evaluated. \( L_{a1} \) still holds since \( a_1 \) (\text{this} in the context of \( a_1 \)) was acquired prior to \( b_1 \) (the one being acquired by \( m3() \)). \( L_{b1} \) is also satisfied since \( T \) is attempting to get a lock on \( b_1 \) and it still does not hold a lock on \( a_2 \), to which \( b_1.a \) is bound. Point 4 does not add any evaluations, since no locks are acquired on the path from point 3. Point 5, however, is quite interesting, since \( T \) is attempting to lock \( a_2 \) on \( a_2.m2() \). \( L_{a1} \) is still satisfied since \( a_1 \) has been acquired prior to \( b_1 \), \( a_2 \) is being acquired at this point (the \text{this} < \( a \) of \( L_a \), with this bound to \( a_1 \), and \( a_2 \) to \( a \)) and \( b_1 \) was acquired prior to \( a_2 \) (the \( b \leq a \) part). \( L_{b1} \) is still satisfied for the same reason as before. \( L_{a2} \), however, is NOT, since \( a_1 \) was acquired prior to \( a_2 \) (the \text{this} < a part of \( L_{a2} \) with this bound to \( a_2 \), and \( a \) to \( a_1 \)). Note that if \( a_2.m1() \) had reassigned the value of member \( a \) to point to another object, say \( a_3 \), \( L_{a2} \) would have been evaluated with a bound to \( a_3 \) and would, then, be satisfied.

It is quite obvious that this is an example of a deadlock-prone system once another thread is introduced, since the threads would attempt to acquire the same locks in different order if they executed \( m1() \) for \( a_1 \) and \( a_2 \) concurrently. The evaluation strategy described above for lock order clauses resembles the idea of axioms for ESC/Java with the exception that it is not prone to the erroneous deadlock warning message mentioned in [31], as discussed in section 3.5. This is due to the fact that the lock order clause is re-evaluated every time a lock acquisition is attempted, solving the inconsistency problems mentioned above. One fine point that should be noted is that there is no need to use the universal quantifier to apply the locking predicate to all instances of a class. The fact that all lock order clauses are evaluated at every lock acquisition point for all instances being manipulated by a thread generates the same effect.
The example above shows how a lock order clause can be used to specify lock acquisition ordering policies and their effect. Specifically, the evaluation at point 5 of La2 illustrates a case in which a lock order clause is violated. One should note that this violation occurs due to the cyclic nature of the example. The interesting point here is that it is not prone to inconsistencies since every acquisition is seen as a potential cause for its violation, so it is re-evaluated in its (attempted) new condition. In summary, the lock order clause allows the local specification of locking policies with global effects. Its simplicity enables sound static and dynamic (RAC) verification of such policies.

5 RELATED WORK

Verification of concurrency properties of programs can be divided into three approaches. Static checking uses the source code only (usually augmented with some annotations) to check the validity of certain properties. Dynamic checking uses only information available during runtime execution of the program under test. There are also approaches that combine both techniques. This work concentrates on dynamic checking.

Lipton’s reduction algorithm [14] allows one to reason about concurrent programs without considering all possible interleavings. The Lockset algorithm was introduced on Eraser [12] to detect race conditions dynamically and extended for object-oriented languages in [14].

Flanagan and Freund [15] describe Atomizer, a dynamic checker for Java programs. It is based on Lipton’s reduction algorithm to define left and right mover operations. It uses the lockset algorithm to keep track of the sharing state of every field an allocated object. Atomizer checks for atomicity, a more 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.” Atomic methods are annotated with the atomic modifier. Atomizer produces instrumentation based on the annotated source code that calls the Atomizer runtime to issue warnings according to the Lockset and reduction algorithms. They report on the use of atomizer on moderate size (up to 90000 lines of
code) programs. They successfully detect errors although the slow down is some cases is considerable (in the order of 40 times).

Stoller et al [17] describe a combination of runtime and static analysis to check for atomicity. They first execute a type discovery task to automatically “annotate” the source code for race-free types. Then they run a static checker to infer the atomicity types. This step produces atomicity warnings. These warnings are then fed to a “focused run-time checking” phase, which checks for races and atomicity violations limited to the portions that caused warnings. They report on checking for races on a system with 157000 lines of code with a speedup of 72% compared with a full run-time checking. They fail to check for atomicity in this system.

Both solutions have a common limitation intrinsic to dynamic checking: they require that the code be exercised to be checked. Not all interleavings are required to be executed, though. Both techniques are able to discover situations in which atomicity is violated even though the particular violation did not happen just by entering an atomic region.

The Extended Static Checker for Java (ESC/Java and ESC/Java2) [31][32] is able to produce warnings regarding possible race conditions and deadlocks by translating specifications into verification conditions and feeding them and the Java source code to a theorem prover. ESC/Java uses axioms to specify lock ordering predicates. Such predicates are required to be valid at the beginning of a method but not at the end. These predicates can be full logical expressions, including quantifiers. This approach has two disadvantages: it is not sound (it does not impact common practical cases, though) and it is not, in general, easily translatable to RAC code.

Nienaltowsky and Meyer [33] present an interesting proposition regarding the use of contracts in a concurrent environment. They target SCOOP [34], 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) on wait conditions that must be eventually satisfied once the current thread acquires a lock on such object. 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. Our proposal reduces to theirs if restricted to the SCOOP model, which specifies that operations on separate objects can only be executed if such objects are arguments of the enclosing routine (the SCOOP runtime is responsible for acquiring the necessary locks prior to executing the routine). This is equivalent as specifying all such objects as \texttt{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. They make a good point in stating that deadlocks can be detected as assertion violations. Our approach treats the problem in a similar way but prevents an actual deadlock from happening by issuing an assertion error immediately upon the violation of a locking policy, i.e. prior to blocking.

Jacobs et al [43][44] present a very interesting approach to the problem of concurrency control of aggregate objects. They present a methodology based on object and thread ownership in which a thread must own an object to access any of its fields. This implies that preconditions and postconditions only refer to thread-safe fields. In other words, the internal behaviour of the object cannot be specified in several important cases. Our approach solves this issue with the introduction of safepoints. When thread-safety of objects referenced by contracts is indeed required, our approach can use the concurrent specification constructs to explicitly state such property so as to allow for its verification.

Without safepoints, all the examples in section 4.1 would have their preconditions reduced to \texttt{true}, i.e. no requirements, since all of them refer to non-thread-safe variables. We see value in being able to specify such properties. The use of safepoints allows the elimination of non-determinism in the specification of a concurrent method. Although a valid specification technique, the use of non-determinism on a RAC scenario defeats the purpose since it would reduce the detail of the specification. The more detail present in a specification, the higher the likelihood of detecting a fault. The techniques for determining atomicity, although very sophisticated, cannot be applied to checking functional properties. Although they can be combined with static techniques that assume
a method to be atomic, thus allowing sequential reasoning to be applied to the verification of functional properties, they cannot be easily adapted to work with RAC. This is due to the fact that interference will always be an issue as long as assertion code is executed without proper protection.

Our work is the first to allow for 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 for concurrent programs in Java-like languages to be completely specified.

6 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 such subscriber uses to connect to the network. The SAE currently supports various devices. This case study focuses 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 such 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 to ensure 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 JUNOS 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, all of which are used in a concurrent environment. Of these, 34 present concurrent behaviour. Table 1 summarises 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 using the thread-safe constructs, i.e. requires_/ensures_thread_safe. The fourth column accounts for the use of locking predicates. All columns show both the total number of methods and the total number of methods that were successfully specified. We consider a specification to be successful if we are able to specify the behaviour the method presents with respect to concurrent aspects. The first line lists the absolute numbers and the second one the percentage of successful specifications in relation to their respective totals.

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

<table>
<thead>
<tr>
<th>Number of methods</th>
<th>Thread-safety spec inheritance</th>
<th>Thread-safe uses</th>
<th>Lock predicate uses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Succ</td>
<td>Total</td>
</tr>
<tr>
<td>Total</td>
<td>347</td>
<td>322</td>
<td>104</td>
</tr>
<tr>
<td>Percentage</td>
<td>100</td>
<td>92.8</td>
<td>30.0</td>
</tr>
</tbody>
</table>

Let us highlight some of the most important results in Table 1. First, 30% of all methods make use of specification inheritance on the concurrent aspect, demonstrating the significance of representing this situation properly. It also shows that our proposed
constructs are able to describe 98.1% of these cases correctly. Second, 23.9% of all methods make use of thread-safety clauses independently of specification inheritance cases, of which 67.5% are correctly specified by the constructs we propose. The cases that could not be specified, 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 behaviour on the concurrent aspect of an object. We are working on extending the thread-safety constructs to accommodate this. Third, this table shows that all cases involving the use of lock predicates were correctly specified by our constructs.

Table 2: Statistics regarding the use of concurrency clauses not related to thread-safety

<table>
<thead>
<tr>
<th>Number of methods</th>
<th>when clause uses</th>
<th>locks_releases clause uses</th>
<th>Safepoints uses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>clause uses</td>
<td>clause uses</td>
<td>Total</td>
</tr>
<tr>
<td>Total</td>
<td>347</td>
<td>10</td>
<td>102</td>
</tr>
<tr>
<td>Percentage</td>
<td>100</td>
<td>2.9</td>
<td>29.4</td>
</tr>
</tbody>
</table>

Table 2 summarizes the uses of other concurrency clauses. The third column displays the total number of methods using the `when` clause. The fourth column displays the total use of the `locks_releases` clause. The fifth column displays the total number of methods that successfully use safepoints. The first line lists the absolute numbers and the second one the percentage relative to the total number of methods. Table 2 shows that the `when` clause despite being used by a small number of methods is required for a complete description of the system. It also shows that the `locks_releases` clause is used by a significant number of methods thus emphasizing its importance. With respect to both clauses, 100% of eligible methods were successfully specified. More importantly, this table shows that 29.1% of all methods required the use of safepoints to be correctly specified and that our proposed construct 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. For instance, the message queue manipulation method
that correlates responses from the router to requests originated on the SAE could not be
specified because it is performed inside a *synchronized* block and such block is
surrounded by observable statements thus forbidding the use of safepoints inside it.

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 cases. 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. Those cases that could not be specified could have been so,
with only one exception, 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.

7 CONCLUSION

Applying Design by Contract to concurrent software poses several challenges, namely
interference, locking properties specification and thread-safety requirements
specification. We tackle interference with the introduction of safepoints, which are
explicit points in the method body at which it is safe to evaluate preconditions,
postconditions and invariants. We define the syntax and semantics of safepoints in the
context of a concurrent method. Based on this concept, we also derive the minimum
thread-safety requirements for a method to be interference-free.

Our approach has a limitation, though. Not every sequential contract can be expressed as
a correct concurrent contract. This is due to the inherent non-determinism of concurrent
systems. This usually happens for contracts that state properties related to sizes of shared
data structures. 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 technique, 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. With such limitation in mind, however, one can design concurrent
contracts that are correct and representative of the actual behaviour of an object in a
concurrent environment.

The current approaches to the specification of thread-safety properties does not
contemplate the impact of specification inheritance since they are placed together with
pre- and postconditions thus being subject to the rules of behavioural subtyping. We
showed this leads to unexpected behaviour. We identify and solve this problem by
creating method specification clauses for thread-safety and locking requirements
analogous to pre- and postconditions with different inheritance rules. We advocate that
such clauses can only be strengthened by specification inheritance, meaning that the
resulting clause is the conjunction of all inherited clauses of the same type, effectively
treating them as invariants. With this approach one can safely specify this type of
properties on interfaces, constraining their implementation to follow them but not
preventing the implementation from requiring and ensuring stronger properties. We
distinguish 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.

We introduce a special construct to the specification of a type, namely the \texttt{lock\_order}
clause, to tackle the issue of lock ordering specification. Its unique semantics solves
previously reported issues with soundness of more general approaches. We feel that,
although less flexible than other approaches, our solution addresses the common needs of
lock acquisition ordering specification with a simple and sound semantics that avoids the
use of quantifiers and logical expressions.
We implemented our proposed constructs on the JML toolset, including the Runtime Assertion Checker. We validated these constructs with a case study consisting of an industry-strength concurrent subsystem that includes 34 concurrent classes and 347 methods. 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.

We still have to explore a way to specify piecewise thread safety of objects that is amenable to inheritance and abstract modeling. Our next step is to continue the work of Briand et al. [7] to determine the effect of the complexity of contracts used as test oracles in the detection of faults with an emphasis on concurrent systems.

8 ACKNOWLEDGEMENTS

This research was supported by Juniper Networks. 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.

9 REFERENCES


SIGPLAN Conference on Programming Language Design and Implementation (PLDI),

Concurrency, Real-Time and Distribution in Eiffel-like Languages (CORDIE), 4-5 July
2006, York, UK.

[34] Arslan V., Eugster P., Nienaltowski P., Vaucouleur S.: "SCOOP - concurrency made easy".

October, 2006.

ECOOP’87, Lecture Notes in Computer Science v. 276, pp. 234-242. Springer-Verlag,
1987.

[37] Pierre America. Designing an Object-Oriented Programming Language with Behavioural
Subtyping. In. Proc. Foundations of Object-Oriented Languages, REX School/Workshop,

[38] Gary T. Leavens and William E. Weihl. Specification and Verification of Object-Oriented

Extensions to Component-Based Systems. In Foundations of Component-Based systems,

Specification Inheritance. In Proc. Of the 18th International Conference on Software

Notes in Computer Science v. 4260, pp. 2-34. Springer-Verlag, November, 2006.

and Modular Reasoning. Department of Computer Science, Iowa State University, TR #06-

[43] Bart Jacobs and Rustan M. Leino and Wolfram Schulte. Verification of Multithreaded
Object-Oriented Programs with Invariants. SAVCBS 2004: Specification and Verification

for aggregate objects with invariants. In proc. IEEE International Conference on Software

[45] C. A. R. Hoare, Monitors: an operating system structuring concept, Communications of the

Variables: Cleanly supporting Abstraction in Design By Contract. Software – Practice and

