Communicating Process Architectures 2015
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2015

A Model-driven Methodology for Generating and Verifying CSP-based Java Code

Julio MARIÑO [a,1] and Raúl N.N. ALBORODO [b,2]
[a] *Babel Group. Universidad Politécnica de Madrid, Spain*
[b] *IMDEA Software Institute, Madrid, Spain*

Abstract. Model-driven methodologies can help developers create more reliable software. In previous work, we have advocated a model-driven approach for the analysis and design of concurrent, safety-critical systems. However, to take full advantage of those techniques, they must be supported by code generation schemes for concrete programming languages. Ideally, this translation should be *traceable*, *automated* and should support *verification* of the generated code. In this paper we consider the problem of generating concurrent Java code from high-level interaction models. Our proposal includes an extension of JML that allows to specify shared resources as Java interfaces, and several translation patterns for (partial) generation of CSP-based Java code. The template code thus obtained is JML-annotated Java using the JCSP library, with proof obligations that help with both traceability and verification. Finally, we present an experiment in code verification using the KeY tool.

Keywords. CSP, JCSP, KeY, Java, JML, shared resources, verification, model-driven, concurrency, message passing

Introduction

Programming concurrent systems is notoriously more difficult and error-prone than programming sequential ones. Reasoning about multi-threaded code is intrinsically complex and, as a consequence of the associated nondeterministic execution, standard software testing techniques cannot provide *sufficient* evidence for correctness. To make things worse, the development of concurrent software is affected by shortcomings in programming language and library support, design methodologies, availability of properly trained practitioners, etc.

In [1], a model-driven approach was proposed as a way of alleviating some of the aforementioned problems. Some benefits of the model-driven development (both in general and, for the specific case of concurrent software) are: 1.- Formalizing (part of) the requirements reduces ambiguity in the problem statement. 2.- Formal models can be the subject of *experiments* aimed at *early requirement validation*. That is, a mathematical model can be formally verified for detecting inconsistencies or other flaws. 3.- Code is not written from scratch but *generated* or *distilled* (semiautomatically) from the model. This brings several benefits. One of them is *portability*. This is specially relevant for concurrent software production, given the volatility of certain languages. A second benefit is robustness against changes in the require-

[1] E-mail: jmarino@fi.upm.es.
[2] E-mail: raul.alborodo@imdea.org.

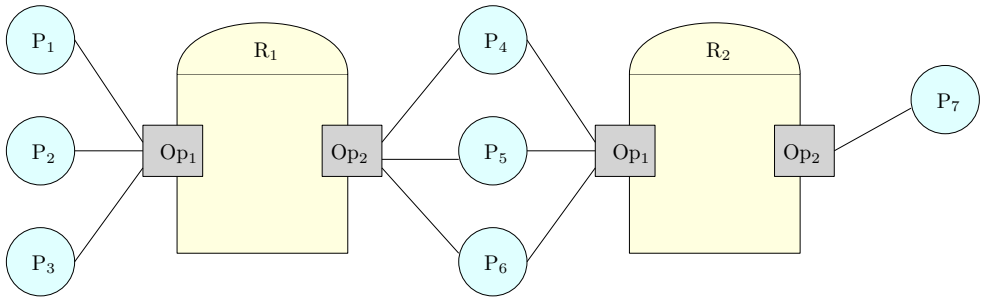

Figure 1. System design based on shared resources: processes are shown as circles, shared resources by the larger squares (with rounded tops) and resource access methods by the smaller squares which are connected to the process.

ments – modifying concurrent code by hand may introduce more errors than re-generating it. Finally, the generative approach may reduce production costs at this stage. 4.- Models can help in the validation, verification and test case generation of the code obtained from the previous phases.

The core idea of our model-driven approach is that the design must distinguish active entities, like clients or processes, from passive ones, such as resources or shared memory locations, as shown in Figure 1. The latter represent all kinds of interaction among the former, and are modelled using an abstraction called *shared resource* that contains a clear interface – which can be invoked from processes – and a transactional transition semantics.

The code obtained from the active entities is considered *light* in the sense that it is assumed to be free from concurrency-specific constructs and is, thus, easier to verify, more portable, and does not require specially trained programmers to develop or test. On the other hand, the code from the shared resources is considered *heavy* code, as it is here where all the concurrency specific code is placed. It is convenient then, that this code is carefully *distilled* from validated designs so that it can be regenerated if requirements change, rather than modified by hand.

The general idea of distilling code from specifications can be approached in different ways. A *fundamentalist* approach is conceivable in which code is automatically obtained from an specification compiler, ensuring correctness by construction and greater maintainability (through code regeneration as changes in specification occur). However, we follow a more realistic view in which the translation is only semi-automatic (schemas for code generation) and the programmer still retains the power of coding or modifying parts of the code for the resource. In this case, it is necessary that the translation is *traceable*, i.e. that the correspondence between code and specification is evident. As the code thus obtained is not automatically correct, there is still the need to verify its conformance to the original specifications, so supporting this verification task is also a crucial requirement for the translation. In [2], two templates for code generation and verification using some of Java's high level mechanisms for shared memory concurrency – *synchronized methods* and *priority monitors* [3] – were proposed along with an extension of JML (*Java Modeling Language* [4]) devised to express shared resources as annotated Java interfaces.

In this paper we complete the aforementioned work by considering the use of (synchronous) message passing mechanisms in Java. Specifically, we describe several templates for generating Java code that uses the JCSP [5] library starting from a shared resource specification. These, as their shared memory counterparts, originated from the classroom use of the model-driven approach in several undergraduate courses on concurrency [6,7]. As said above, the method is semi-automatic in the sense that programmers still retain the power of coding

part of the shared resource implementation and, in doing so, they can also introduce errors. In order to ensure a proper application of the method we propose a set of proof obligations that must be checked at certain points of the code. Finally, we show how these properties can be checked using an existing program verification tool (KeY [8]). It is important to note that KeY knows nothing about JCSP channels nor any other synchronization/communication mechanism. In fact, KeY is a verification system for single threaded Java code that knows nothing about concurrency. In order to use it as a tool for verifying concurrent code some *tricks* are needed, such as taking advantage of the code structure – which is fixed by the template employed – or mocking part of the semantics of channels, etc., so that KeY can make some inferences required by the proof obligations.

The paper is organized as follows: next section introduces the notation used to specify shared resources by example. Schemata for code generation and their associated proof obligations follow in section 2. Section 3 focuses on the experimental part where we present the code instrumentation and its verification using the KeY verification system. Finally, section 5 summarizes results, compares with related approaches and describes future work.

1. Shared Resources

Figure 2 shows a formal specification of a shared resource using the notation introduced in [6]. This is the well-known *readers and writers* synchronization pattern. Readers are expected to follow a protocol in which *beforeRead* is invoked first, then some shared database is accessed, and finally *afterRead* is invoked. Writers follow a similar protocol involving *beforeWrite* and *afterWrite*. The OPERATIONS section specifies the public interface of the shared resource. Notice that all operations must follow a transactional semantics: the POST clauses specify the effect of executing a single operation, and any interleaving of them must be equivalent to some sequential interleaving.Variables with *in* as superscript denote their initial value while those without annotations refer the output – i.e. return-time – values.

Condition synchronization is specified by the CPRE clauses: every state change involving some operation is performed only if the concurrency precondition holds. Classical preconditions (denoted PRE) can also be specified to rule out illegal calls to the resource operations, such as invocations with invalid parameters or those which violate a protocol known to the caller (e.g. that `afterWrite` follows, and only follows, a `beforeWrite`).

Although the mathematical notation has the virtue of being fully language-independent, sometimes, it is practical to have a way of dealing with language-dependent issues that will appear during the code generation phase: constants, modules, initialization, parameters, etc., to name a few. For these reasons, in [2] we proposed to specify shared resources as Java interfaces annotated with JML. Conceptually, this agrees with the view that Java interfaces are meant to express a contract for potential implementations. In the case of a shared resource, this means that different classes can implement the same resource, e.g. using disparate synchronization models, such as shared memory or message passing. Additional advantages of this approach is that the logic annotations sit reasonably well alongside JML. Indeed, only two basic clauses had to be added to the original JML syntax:

- **shared_resource** declares that all instances of the class represent a shared resource. This is a quite strong requirement on classes, as it implies the transactional semantics of interleavings of method calls.

CADT ReadersWriters

OPERATIONS
ACTION BeforeRead:
ACTION AfterRead:
ACTION BeforeWrite:
ACTION AfterWrite:

SEMANTICS
DOMAIN:
STATE: $(readers : \mathbb{N} \times writers : \mathbb{N})$
INVARIANT: $(readers > 0 \Rightarrow writers = 0) \wedge$
$(writers > 0 \Rightarrow readers = 0 \wedge writers = 1)$
INITIAL: $writers = 0 \wedge readers = 0$

CPRE: $writers = 0 \wedge readers = 0$
BeforeWrite
POST: $writers = 1$

PRE: $writers = 1$
CPRE: *true*
AfterWrite
POST: $writers = 0$

CPRE: $writers = 0$
BeforeRead
POST: $readers = 1 + readers^{\text{in}}$

PRE: $readers > 0$
CPRE: *true*
AfterRead
POST: $readers = readers^{\text{in}} - 1$

Figure 2. Readers/Writers specification according to the notation introduced in [1]. CADT is an acronym for *Concurrent Abstract Data Type* and represents a *shared resource*. CPRE represents a *concurrent* (or *synchronization*) *pre-condition*, meaning that invocations of the operation are allowed but will be blocked if the CPRE is false. PRE means the operation should not be invoked if the PRE condition is false. Invokers are expected to know the state of that pre-condition. They are not expected to know the state of the CPRE, which depends on the state of the shared resource.

- **cond_sync** declares that any invoker will be blocked when the associate predicate does not hold and should be in this state until the condition becomes true by the changing of the shared resource inner state. In other words, this corresponds to the CPRE clauses in the mathematical notation.

The rest of clauses in the shared resource specification – PRE, POST, INVARIANT, INITIAL, etc. – can be cast into standard JML declarations which means that our extension should not be very disruptive for JML practitioners. The resulting Java interface for the readers & writers example is shown in Figure 3.

Formal specifications and their associated source code, in addition to several examples of varying complexity can also be found at [9]. In more realistic examples than the ones considered in this paper, process interaction is not limited to a centralized control and processes may interact via several shared resources. However, this poses no problem for the techniques presented in this paper, as here we are just concerned with how the code implementing a

```
package es.upm.babel.ccjml.samples.readerswriters.java;

public interface /*@ shared_resource @*/ ReadersWriters {
  //@ public model instance int readers;
  //@ public model instance int writers;

  /*@ public instance invariant
    @    readers >= 0 && writers >= 0 &&
    @    (readers > 0 ==> writers == 0) &&
    @    (writers > 0 ==> readers == 0 && writers == 1);
    @*/

  //@ public initially readers == 0 && writers == 0;

  /*@ public normal_behaviour
    @   cond_sync writers == 0 && readers == 0;
    @   assignable writers;
    @   ensures writers == 1;
    @*/
  public void beforeWrite();

  /*@ public normal_behaviour
    @   requires writers == 1;
    @   assignable writers;
    @   ensures writers == 0;
    @*/
  public void afterWrite();

  /*@ public normal_behaviour
    @   cond_sync writers == 0;
    @   assignable readers;
    @   ensures readers == \old(readers) + 1;
    @*/
  public void beforeRead();

  /*@ public normal_behaviour
    @   requires readers > 0;
    @   assignable readers;
    @   ensures readers == \old(readers) - 1;
    @*/
  public void afterRead();
}
```

Figure 3. Readers/Writers Java interface with JML annotations.

single resource conforms to its original specification. This is what we sometimes call ”internal verification”. Correctness of the whole system is studied at the model level analysing the properties of the transition system generated by the processes and the shared resources they interact with. We sometimes refer to this as ”external verification”. One of the advantages of the model-based approach is indeed this separation of concerns that allows to keep language dependence to a minimum.

2. Model-based Code Generation Using JCSP

The following paragraphs present two templates for translating a shared resource specification into a Java class implementation. For each proposal, in addition to specifying the translation scheme, special attention is paid to the implicit proof obligations required to ensure that the intended behaviour of the specification is preserved. In order to illustrate the applicability of these templates, translations of the *Readers/Writers* (Figure 2) and *Multibuffer* (Figure 7) specifications are provided.

The core idea of the translation is to identify the shared resource with a *server process* that manages the resource's internal state in its local memory and deals with the requests of the other (design level) threads in a client/server fashion. In order to avoid contamination of the clients' code we advocate the creation of a wrapper that hides the actual implementation of the resource. That is, a JCSP-based implementation of a given resource will usually implement both the *CSProcess* interface – so that the public `run()` method implements the server – and the interface where the specification of the shared resource resides – which will force the implementor to provide definitions for the resource operations. However, the implementations of the resource operations will only consist in message-passing operations to and from the server, which does the "real" job, resulting in a kind of *remote procedure call* style.

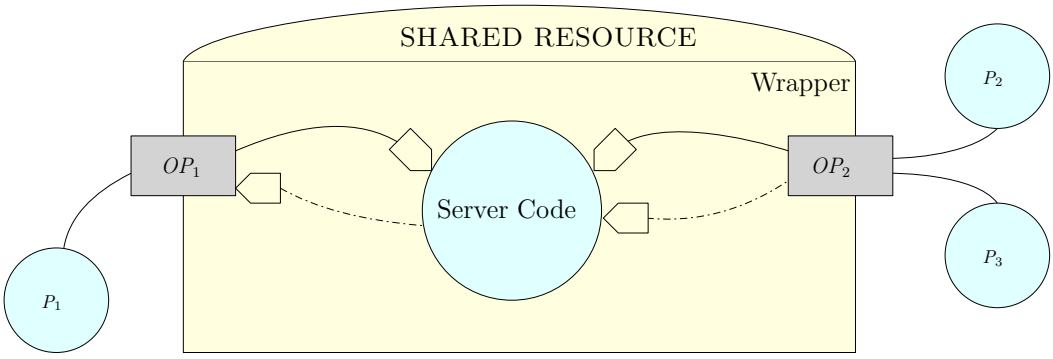

Figure 4. Graphical representation of our message passing approach: the wavy lines indicate channels permanently present; the dotted lines indicate channels created on demand by an operation method, if needed, and only used during the operation.

This architecture is shown graphically in Figure 4. That is, clients of the shared resource cannot distinguish between an implementation using CSP or another concurrency mechanism. Summarizing, the implementation consists of the two aforementioned components whose responsibilities are:

Wrapper: Receiving method invocations and propagating them as messages to the server through CSP channels. Also, if the operation returns a value, waiting for the server to produce such a value and read it from a channel (which may have been created at method invocation time).

Server: Processing the requests received from the wrapper methods and modifying the shared resource inner state according to methods POST specification. Of course, requests should only be served if the CPRE is satisfied.

Applying these ideas to a simple resource like the Readers/Writers presented in the previous section is a fairly easy task. None of the four operations take arguments nor return anything, so their implementation consists in sending out a meaningless token (a **`null`**). A channel for each operation works fine. For the `after` operations, the CPRE is *true*, so the channel will be always available and the operation will take virtually no time from a client's perspective. For the `before` operations, the CPRE is a formula which only depends on the resource's internal state, which makes it possible to enable or disable the corresponding channel using the conditional reception mechanism in CSP. This is implemented in JCSP by creating an object of the `Alternative` class from a four-channel array, and then invoking the `fairSelect(conds)` method on that object, where *conds* encodes the corresponding CPREs. The resulting code is shown in Figure 5.

```
1  public class ReadersWritersCSP implements CSProcess,ReadersWriters {
2
3    protected final Any2OneChannel chBeforeWrite = Channel.any2one();
4    protected final Any2OneChannel chBeforeRead  = Channel.any2one();
5    protected final Any2OneChannel chAfterWrite  = Channel.any2one();
6    protected final Any2OneChannel chAfterRead   = Channel.any2one();
7
8    protected int readers = 0;
9    protected int writers = 0;
10
11   // interface ReadersWriters
12   public void beforeWrite() {              // No PRE condition
13     chBeforeWrite.out().write(null);
14   }
15   public void afterWrite() {
16     //@ assume writers == 1;
17     chAfterWrite.out().write(null);
18   }
19   public void beforeRead() {               // No PRE condition
20     chBeforeRead.out().write(null);
21   }
22   public void afterRead() {
23     //@ assume readers > 0;
24     chAfterRead.out().write(null);
25   }
26
27   // the server
28   public void run() {
29     // One entry for each method
30     Guard[] guards = {chBeforeWrite.in(), chBeforeRead.in(),
31                       chAfterWrite.in(),  chAfterRead.in()};
32     final int BEFORE_WRITE = 0, BEFORE_READ = 1,
33               AFTER_WRITE = 2, AFTER_READ=3;
34     // conditional reception
35     boolean syncCond[] = new boolean[4];
36     Alternative services = new Alternative(guards);
37     while (true) {
38       syncCond[BEFORE_WRITE] = (readers == 0 && writers == 0);
39       syncCond[BEFORE_READ]  = writers == 0;
40       syncCond[AFTER_WRITE] = true;
41       syncCond[AFTER_READ]  = true;
42       int chosenService = services.fairSelect(syncCond);
43       switch(chosenService){
44         case BEFORE_WRITE:
45           chBeforeWrite.in().read();
46           //@ assert writers == 0 && readers == 0;
47           writers++;
48           break;
49         case BEFORE_READ:
50           chBeforeRead.in().read();
51           //@ assert writers == 0;
52           readers++;
53           break;
54         case AFTER_WRITE:                  // no CPRE condition
55           chAfterWrite.in().read();
56           writers--;
57           break;
58         case AFTER_READ:                   // no CPRE condition
59           chAfterRead.in().read();
60           readers--;
61           break;
62       }
63     } // end while
64   } // end run
65 }
```

Figure 5. Readers/Writers implemented using JCSP.

The structure of the service loop inside the `run()` method is quite simple. The `syncCond` array with the reception conditions is computed first and then, depending on the index value returned by `fairselect(syncCond)`, the code for the selected operation is executed. Observe that the code that evaluates `syncCond` has been slightly optimized to avoid reassigning positions with constant values. In the present example this is trivial but, in general, optimizing this part of the code trying to avoid recomputing positions that have not changed since the previous iteration is one of the most common sources of mistakes.

Given the structure of the code, a set of proof obligations can be imposed at certain places. These will set the basis for the verification techniques described in Section 3:

- The *preconditions* (PREs) are often independent from state changes in the resource (e.g. those in Figure 7, but not those in Figure 2). In such cases, they should be checked at call time – that is, inside the wrapper code – and throw the corresponding exception although, for brevity, we have not included that in these examples. In any case, they must be assumed before the relevant operations (lines 16, 23 of Figure 5).
- The *invariant* (INVARIANT) maps to the loop invariant within the server code. This is because the structure of this code serialises the operations (refusing them until their CPREs hold), performing one (and only one) in each loop – and no state changes happen between the start of the loop and the start of an operation. Therefore, the proof obligation regarding the invariant is to establish it before first entry to the loop (line 37) and that it is restored by the end of the loop (line 63). Again because of the logic in the server code, the end of the loop is the **`break`** at the end of each operation (lines 48, 53, 57, 61).
- The *concurrent or synchronization pre-condition* (CPRE) must hold right before entering the code for each operation (lines 46, 51).
- The *post-condition* (POST) must hold on exit of the code of each operation (lines 48, 53, 57, 61).

However, in general, such a simple scheme is not possible. Typically, resource operations have parameters and return values and some of their CPREs may depend on those parameters. A more sophisticated protocol is then needed for the server to decide when a request may be processed. Figure 6 shows the different kinds of CPRE and how they can be systematically treated. The two styles described in Sections 2.1 and 2.2 develop these ideas.

$$\text{CPRE}(op_i(\vec{x},\vec{y})) \equiv C_i \begin{cases} \textit{tautology} & C_i \Leftrightarrow \textit{true} \quad \textit{open channel} \\ \textit{depends only on resource state} & C_i = \phi(S) \quad \textit{one channel enabled by } \phi \\ \textit{may depend on } \vec{x} : C_i = \phi(S,\vec{x}) & \begin{cases} \textit{channel replication} \\ \textit{deferred requests} \end{cases} \end{cases}$$

Figure 6. Mapping CPREs to channels according to their logical analysis.

2.1. Channel Replication

When an operation's CPRE depends on some of its parameters, the server cannot receive the request for that operation on a single channel and decide whether it addresses the request based on data it has not received yet. Indeed the same channel should be open for some requests and closed for others *at the same time*. For example, consider the *Multibuffer* specification given in Figure 7. Both CPREs depend on values obtained from their parameters: a nearly empty buffer could process a small *Get* request but not a large one; a nearly full buffer could process a small *Put* but not a large one[1].

CADT Multibuffer

OPERATIONS
ACTION Put: $Sequence(ANY)[i]$
ACTION Get: $\mathbb{N}[i] \times Sequence(ANY)[o]$

SEMANTICS
DOMAIN:
STATE: $\text{self} = Sequence(ANY)$
INVARIANT: $\mathsf{Length}(\text{self}) \leq MAX$
INITIAL: $\mathsf{Length}(\text{self}) = 0$

PRE: $1 \leq \mathsf{Length}(r) \leq \lfloor MAX/2 \rfloor$
CPRE: $1 \leq \mathsf{Length}(r) \leq MAX - \mathsf{Length}(\text{self})$
Put(r)
POST: $\text{self} = \text{self}^{\text{in}} + r$

PRE: $1 \leq n \leq \lfloor MAX/2 \rfloor$
CPRE: $1 \leq n \leq \mathsf{Length}(\text{self})$
Get(n, s)
POST: $\text{self}^{\text{in}} = \text{self} + s$

Figure 7. Multibuffer specification according to the notation introduced in [1].

The first solution we will consider is to drop the assumption that every operation has a single dedicated channel to the server. In fact, if we manage to use so many channels that requests for implementing some operation never coincide on the same channel when their CPREs evaluate to different boolean values, then the problem is solved. Systematically doing this corresponds to the technique we have named *channel replication*.

Consider, for instance one operation op_i that takes arguments x and y so that its CPRE C_i depends on x but not on y. Formally, let D_x, D_y be the domains for x and y. C_i is independent from y iff $\forall a \in D_x . \forall b, b' \in D_y . C_i[a/x, b/y] \Leftrightarrow C_i[a/x, b'/y]$. On the other hand, C_i depends on x iff $\exists a, a' \in D_x . C_i[a/x] \not\Leftrightarrow C_i[a'/x]$. C_i partitions D_x into a set of equivalence classes trivially: $a, a' \in D_x$ are equivalent iff $C_i[a/x]$ and $C_i[a'/x]$ are logically equivalent. Let E_i be the set of such equivalence classes. If E_i is finite, then we can assign one channel to each equivalence class. This way, we ensure that two requests serving $op_i(a,b)$ and $op_i(a',b)$ will only be routed to the same channel if the precondition holds (or fails) for them both, as required above.

[1] During the verification we assume the consistency of the Multibuffer specification. The `MAX/2` limit guarantees that, at a given (resource) state, only putters or getters can be blocked, but not both. This enforces deadlock freedom of a system with client processes which repeatedly try to send and read sequences of arbitrary length. In [1] the property is mechanically proven (for a fixed number of clients) using a model checker.

Of course, these channels will have to be placed somewhere inside the overall array of channels of the server process. As E_i is finite, there must be some (partial) injective function $pe_i : E_i \nrightarrow \mathbb{N}$. By the construction of E_i we also assume the existence of another (total) injection $px_i : D_x \rightarrow E_i$. That means that in order to route a call to $op_i(a,b)$, the wrapper method will have to send a request over the channel that occupies position $pe_i(px_i(a))$ from the server's point of view. The server is responsible for enabling that channel on the condition $C_i[a/x]$ – the a of choice is immaterial, as the value is the same for the whole equivalence class.

If the same technique is applied on another operation op_j, it must be ensured that the ranges of pe_i and pe_j are disjoint. This will allow the server to identify the operation from the index returned by the `fairSelect` method. As a final remark, if some operation depends on several parameters, the method can be adapted using products as long as the finiteness of the equivalence classes condition holds for each parameter. Also, observe that the case of CPREs independent of parameters can be seen as a degenerate case where there is only one class of equivalence.

2.1.1. Multibuffer Implementation (by Replicated Channels)

Figure 8 outlines an implementation of the multibuffer resource (Figure 7) using the channel replication technique.

```
16 public class MultibufferCSP implements CSProcess {
     ...
18   protected final int MAX;
19   protected Object[] buffer;
20   protected int nData = 0; // number of items of the buffer
     ...
22   /** WRAPPER IMPLEMENTATION */
23   private final Any2OneChannel[] chPut;
24   private final Any2OneChannel[] chGet;
     ...
30   public MultibufferCSP(int length) {
31     MAX = length;
32     chPut = Channel.any2OneArray(MAX/2);
33     chGet = Channel.any2OneArray(MAX/2);
34     buffer = new Object[MAX];
       ...
37   }
38
39   public void put(Object[] objs) {
40     chPut[objs.length - 1].out().write(objs);
41   }
42
43   public Object[] get(int n) {
44     Object[] result;
45     One2OneChannel chResp = Channel.one2one();
46     chGet[n - 1].out().write(chResp.out());
47     result = (Object[]) chResp.in().read();
48     return result;
49   }
```

Figure 8. Multibuffer implementation using the channel replication technique. Channel definition based on the domain of the input parameters. Wrapper (client side) operations.

The CPRE of the `put` operation depends on a parameter `objs` (a sequence of objects) that can take infinitely many values, i.e. D_{objs} is infinite. However, this CPRE depends only on its length, so the set of equivalence classes, E_{objs}, will have cardinality `MAX/2`. Therefore,

put can be dealt with by sending its message down an appropriately chosen element from an array of channels (chPut): the choice index being one less that its length (because zero length puts are disallowed). When the server is deliberating over incoming channel selection, the size of the incoming message is implicit from the index of the channel. Hence, requests that it cannot currently handle can be refused (i.e. blocked).

The CPRE of the get operation depends on a parameter n, whose domain is a small range of integers 1..MAX/2. Therefore, these operations can be handled similarly to the puts, by sending a message down one of a different array of channels (chGet). The message sent is the output-end of a newly made channel, chResp, for the resource to use (when it can) to return the result. The get operation simply waits for it.

```
60    public void run() {
        ...
70      Guard[] inputs = new Guard[MAX];
71      for (int n = 0; n < MAX/2; n++) {
72        inputs[n] = chPut[n].in();
73        inputs[n + MAX/2] = chGet[n].in();
74      }
        ...
76      final Alternative services = new Alternative(inputs);
        ...
84      boolean[] syncCond = new boolean[MAX];
85      while (true) {
          ...
88        for (int n = 0; n < MAX/2; n++) {
89          syncCond[n] = MAX - nData >= n+1;   // put CPRE
90          syncCond[n + MAX/2] = n+1 <= nData; // get CPRE
91        }
          ...
100       int chosenService = services.fairSelect(syncCond);
          ...
105       if (chosenService < MAX/2) {   // put
106         int putSize = chosenService + 1;
107         //@ assert MAX - nData >= putSize;  // CPRE Put
            ... input putSize objects from chPut[putSize -1]
            ... put them in the buffer
            nData += putSize;
110       } else {  // get
111         int getSize = (chosenService - MAX/2) + 1;
112         //@ assert getSize <= nData;        // CPRE Get
            ... input response channel-end from chGet[getSize - 1]
            ... remove getSize objects from buffer
            nData -= getSize;
            ... output the removed items down the response channel
115       }
116     }// end of server loop
117   } //  end of run method
```

Figure 9. Multibuffer implementation using the channel replication technique. Server component code.

Figure 9 outlines an implementation of the process servicing all the requests. The channels are put into a Guard array (lines 70-74) according to the pe_i functions:

$$pe_{put}(n) = n$$
$$pe_{get}(n) = MAX/2 + n$$

The JCSP Alternative is constructed with these guards (line 76), which will be fairSelected using the syncCond array of pre-conditions (line 100). Since these pre-

conditions depend on the number of items in the buffer, they are computed at the start of each service loop (lines 88-91).

The pe_i functions were designed to have disjoint ranges which ensures the existence of an inverse mapping from $\mathbb{N}$ to $\{put, get\}$. This is used, first, to determine the original operation from the value returned by `fairSelect` (lines 105 and 110) and, then, to recover the original parameters by taking the inverses px_{put}^{-1} and px_{get}^{-1} (lines 106 and 111).

2.2. Deferred Requests

The previous schema requires that the set of equivalence classes of all the calls to some resource operation, modulo logical equivalence of their CPREs, is finite and, for it to be practical, relatively "small". Also, it can be argued that the interplay of mappings ($pe_i, px_j \ldots$) involved is error-prone. An alternative schema that does not suffer from these problems is what we call *deferred requests*. As we will see, the only (quite reasonable) requirement is that the number of clients (for some operation) is finite.

The main idea is to keep one single channel per operation. As explained before, this implies that if an operation CPRE depends on a parameter, the condition cannot be encoded in the corresponding reception list. A solution for this is to keep method channel open but transforming each simple message as a *request*. It must contain a *footprint* of the parameters (values needed to compute the CPRE) and a fresh channel used either to keep the client blocked until the operation has been performed (in the case that no results are to be returned) or for information exchange.

2.2.1. Multibuffer Implementation (by Deferred Requests)

An implementation of the *wrapper* operations for the multibuffer example using this technique is shown in Figure 10. The message sent consists of a newly constructed channel (lines 46 and 54) bundled together with a *footprint* of the method parameters. The footprint contains sufficient information for the server to compute the relevant CPRE. Depending on the content of the footprint sent, two cases can be distinguished:

- *single send*: when the footprint contains all the actual parameter (e.g. the `get` operation, line 55), the method call can proceed without sending any further information.
- *double send*: when the footprint does not contain all the parameter information (e.g. the `put` operation, line 47), the rest of the formal parameter must be sent through the already created fresh channel (line 49).

Notice that in both cases, the client will block its execution until a message (the return value or a token) is written into the invocation associated channel notifying that the request was processed successfully (lines 50 and 56 of Figure 10).

Nevertheless, the tricky part comes when designing the server code (Figure 11). This is a sequential process that defers all requests into some auxiliary data structure, to be processed later. In order to avoid errors and to obtain a correct implementation of this component, a set of considerations must be taken into account:

- Every request is stored in some data structure as soon as it is received by the server. Typically, there will be one collection per method.
- It must be ensured that no pending request whose synchronization condition holds is left unattended before entering into a new iteration of the service loop.
- Finally, mutual exclusion of the servicing of the requests must be guaranteed by the server implementation.

```
19 public class MultibufferCSPDeferredRequest implements CSProcess {
     ...
21   protected final int MAX;
22   protected Object[] buffer;
23   protected int nData = 0; // number of items of the buffer
     ...
25   /** WRAPPER IMPLEMENTATION */
26   private final Any2OneChannel chPut = Channel.any2one();
27   private final Any2OneChannel chGet = Channel.any2one();
     ...
38   public MultibufferCSPDeferredRequest(int length) {
39     MAX = length;
40     buffer = new Object[length];
       ...
43   }
     ...
45   public void put(Object[] els) {
46     One2OneChannel innerChannel = Channel.one2one();
47     chPut.out().write(new PutRequestCSP(els.length,innerChannel));
48     // send the data to be inserted
49     innerChannel.out().write(els); // blocks until server can take it
50     innerChannel.in().read();      // wait for server to finish
51   }
52
53   public Object[] get(int n) {
54     One2OneChannel innerChannel = Channel.one2one();
55     chGet.out().write(new GetRequestCSP(n,innerChannel));
56     Object[] res = (Object[]) innerChannel.in().read();
57     return res;
58   }
```

Figure 10. Multibuffer implementation using the deferred requests technique. Only two channels. Wrapper (client side) operations.

There are several implications associated to the first point. Unlike channel replication, requests are no longer served in the same order as they are processed by the server, so the responsibility for ensuring fairness – or any other priority policy – falls on the programmer. Of course, the data structures chosen to store pending requests must facilitate searching for the requests whose synchronization precondition holds.

Regarding the other points, if some enabled request is left unattended and the server invokes `fairSelect`, this may force a client block unnecessarily, with the resulting loss of concurrency. Finally, mutual exclusion is guaranteed as long as the server is purely sequential code (processing each request completely without being interleaved with another one).

Figure 11 outlines the key details of the server code for multibuffer example. An invariant at the start of the main loop (lines 73-134) is that none of the requests held in the lists (`producersRequest` and `consumersRequest`) can currently be processed – because their CPREs are all *false*. On first entry, this is trivial since the lists are empty. For subsequent entries, the (JML) asserions at the end of the loop show that the invariant is re-established.

The first action of the main loop, therefore, is to wait for a new request (line 76) and add it to the appropriate list (lines 77-86). Following this, there is an inner loop (lines 88-122) in which the lists are re-examined, possibly many times, to see if any held requests can now be processed and, if so, process them and remove them from the lists.

Within that inner loop, there are two further loops: the first (lines 91-107) processes deferred producer requests, where possible, and the second (lines 108-121) does the same for consumer requests. Usually, processing these lists of deferred requests just once will leave

```
66    public void run() {
        ... construct lists, producersRequest and consumersRequest,
        ... to hold all client requests for subsequent processing
69      /* One entry for each method. */
70      Guard[] inputs = {chPut.in(), chGet.in()};
71      Alternative services = new Alternative(inputs);
72
73      while (true){
74        // wait for new request and add it to appropriate list
75        // (synchronization conditions are not checked here)
76        int chosenService = services.fairSelect();
77        switch(chosenService){
78          case PUT:
79            PutRequestCSP pRequest = (PutRequestCSP)chPut.in().read();
80            producersRequest.add(pRequest);
81            break;
82          case GET:
83            GetRequestCSP gRequest = (GetRequestCSP)chGet.in().read();
84            consumersRequest.add(gRequest);
85            break;
86        }
87
88        boolean anyResumed;
89        do {
90          anyResumed = false;
91          // process deferred producer requests (where possible)
92          int lastItem = producersRequest.size();
93          for (int i = 0; i < lastItem; i++) {
              ... dequeue PutRequestCSP item from producersRequest
              ... extract the put footprint, putSize, from the item
97            if (putSize <= MAX - nData){  // CPRE put
98              //@ assert MAX - nData >= putSize;
                ... extract the channel, innerChannel, from the item
                ... input putSize objects from innerChannel
                ... put them in the buffer
                nData += putSize;
                ... send a null back down innerChannel
                anyResumed = true;
              } else {
                ... enqueue item back in producersRequest
106           }
107         }
108         // process deferred consumer requests (where possible)
109         lastItem = consumersRequest.size();
110         for (int i = 0; i < lastItem; i++) {
              ... dequeue GetRequestCSP item from consumersRequest
              ... extract the get footprint, getSize, from the item
114           if (getSize <= nData){  // CPRE get
115             //@ assert getSize <= nData;
                ... remove getSize objects from buffer
                nData -= getSize;
                ... extract the channel, innerChannel, from the item
                ... output the removed items down innerChannel
                anyResumed = true;
              } else {
                ... enqueue item back in consumersRequest
              }
121         }
122       } while (anyResumed);
          ... JML asserts that the CPREs for all the items left in
          ... producersRequest and consumersRequest are false
134     }
135   }
```

Figure 11. Multibuffer implementation using deferred requests. Server side message processing.

none that can still proceed. But, in some cases, there can be a *'ping pong'* effect between several operations. That is, serving a request from one operation can *enable* another request that was already checked and deferred again (because its CPRE did not hold with the state at that time).

For a simple example, drop the PREs from the specifications and consider the scenario with (i) `get(2)`, (ii) `put([a,b])` and (iii) `put([c,d,e])` as deferred requests and an empty buffer of 3 elements. The first loop (dealing with `put` requests) will process petition (ii) (there are 3 free slots, more that needed) leaving 2 filled and 1 empty, so (iii) cannot be processed and will be deferred. The second loop (dealing with the `get` requests) will process (i) leaving a buffer with 3 empty slots. After that loop, the state of the server does not meet the second of the three considerations given above for correct implementation – because (iii)'s CPRE now holds and has not been processed.

This is why these two (innermost) request processing loops are within the middle loop (lines 88-122) that exits only when none of their CPREs are true, handled through the `anyResumed` flag. For verification (Section 3.2.1), `asserts` to that effect are made following that enclosing loop (re-establishing the main loop invariant mentioned earlier).

3. Verification

We present a framework for verifying the code obtained using either of the aforementioned templates using an existing deductive verification tool. We have chosen KeY [8] because it supports Java code with JML annotations and is continuously maintained offering a solid verification framework that suites perfectly the purposes of this paper. Other systems examined – and discarded for various reasons – were Bogor [10] and TACO [11].

KeY is a dynamic logic verification platform for JavaCard code which enables proof correctness of a given behavior specification annotated in JML, using contract based techniques for verification. These specifications include precondition P (*requires* JML clause), postcondition Q (*ensures* clause) and method modifiers (*assignable* or *modifiable* clause). KeY tries to prove that the execution of an operation, in a state in which class invariant I and the precondition P holds, will terminate normally and in a state in which the postcondition holds.

To accomplish the verification, we start with code obtained from specifications (Figures 2 and 7) by application of one of the techniques described in Section 2. This code is expected to contain logical (JML) annotations at certain points determined by the template logic. Then, the code (Java and JML) needs to be instrumented for KeY to be able to reason about the concurrent interplay between server and clients. This instrumentation helps KeY figure out what happens in the client side from the state of certain data structures local to the server that represent the behavior of channels, etc.

The goal of this verification process is just to establish the conformance of the implementation with the template applied and, consequently, that the implementation behaves as expected with respect to the state transition semantics of the shared resource specification. The correctness of the whole system needs to be established in some other way, for instance by reasoning about the transition system generated by clients and shared resources – e.g. using model checking techniques, as shown in [1]. This system level validation is, however, language-independent, what means that dealing with code issues is circumscribed to the first part of the problem. This is specially convenient given that tools for reasoning about state machines are very powerful, while support for program verification is rather unequal and often restricted to small (and sequential) subsets of certain programming languages.

The following paragraphs present the verification process using a common structure. First, we enumerate the proof obligations that the templates impose at relevant points of the program. Then, we present how to instrument the generated code in order to make it verifiable by KeY. Finally, we give details on actually proving the properties in KeY.

3.1. Directly Processed Requests

The implementation of the *Readers/Writers* resource (Figure 5) and the channel replication implementation of the *Multibuffer* (Figures 8 and 9) enable the respective servers to determine whether the relevant CPREs hold for all the operation request channels they are servicing, before taking any request. Hence, requests for service are blocked (if the CPRE is false) or accepted and immediately processed (if the CPRE is true). This is not the case for the deferred request implementation of *Multibuffer* (Figures 10 and 11) and verification of such implementations is presented in Section 3.2.

This subsection considers verification of servers that can block currently unserviceable requests or process serviceable ones immediately. In the code fragments illustrating this, we show the discovery of the particular request selected for servicing via Java's `switch`/`case` structure (used by the Readers/Writers server), rather than `if`/`else` (used by the Multibuffer server via channel replication). Modifying this generic template for the `if`/`else` structure is straightforward.

3.1.1. Proof Obligations

Let us start by proposing a set of proof obligations for the server component of our first code generation template:

- *prop_cs_preservation:* immediately after the conditional statement that decides upon the index that tells the server which call must serve, the CPRE of that call must hold. As can be seen in Figure 9, it is asserted that CPRE holds just before updating the shared resource (line 107 for `put` operation and line 112 for `get`). This restriction ensures *safety* of the server, i.e. no request is served in a state that does not satisfy its synchronization condition.
- *prop_safe_selection:* the server code must guarantee that a valid service is selected in each iteration, i.e. the selected service *s* must belong to *pe* range, and has a message waiting to be read. This property is strictly related to `syncCond` array structure because `fairSelect` will chose a service depending on the available channels expressed by the aforementioned array.
- *prop_only_one_request:* only one request is processed in each iteration. Server code must guarantee this in order to avoid losing requests.

3.1.2. Instrumentation

As concurrency in general, and the JCSP library in particular, are not supported natively by KeY, code instrumentation for channels is needed. They are translated into either an **int** variable representing the length of the message queue (when the contents of the request are immaterial for the subsequent verification) or into a `List` (when some operation makes use of the request contents and that makes it it necessary to keep a sequence of request blueprints so as to verify the correctness of the code implementing that operation.) Moreover, we need to simulate the behavior of `fairSelect` via the homonym method shown below:

```
//@ requires (syncCond.length == guards.length);
/*@ requires (\forall int j; 0<=j && j<guards.length;
  @                          guards[j]!= null);
/*@ ensures (\result >= 0 && \result < syncCond.length &&
  @          syncCond[\result] && guards[\result].size() > 0 )
  @         ||
  @         (\result == -1 &&
  @           (\forall int i; i >= 0 && i < syncCond.length;
  @                           !syncCond[i] || guards[i].size() == 0)
  @         )
  @*/;
private int /*@ pure @*/ fairSelect(
  boolean[] syncCond, List[] guards
){
  for(int i = 0 ; i < syncCond.length; i++){
    if (syncCond[i] && guards[i].size() > 0)
      return i;
  }
  return -1;
}
```

This tells us that we are able to select a channel if and only if there is a message waiting on it and its associated synchronization condition holds.

An advantage of KeY is the possibility of using of *contracts* (with the *Method treatment* option by selecting *Contract*) that make it simpler to simulate JCSP `fairSelect` method. To fully express its behavior, when there is no "selectable" channel (shared resource is in a state in which no CPRE holds), -1 is returned to halt the server loop. Because of this definition, the server loop condition is changed to iterate until there is no request that can be processed, as shown in Figure 12.

```
int chosenService = 42;   // anything but -1
int[] services = {...};
boolean[] syncCond = new boolean[#rg(pe)];

while (chosenService != -1 ){
  ...  update syncCond array

  /*@ assert (\forall int j; 0<=j && j<syncCond.length;
    @                        syncCond[j] == CPRE_j);
    @*/

  chosenService = fairSelect(syncCond, services);

  //@ assume chosenService < guards.length && chosenService >= -1;
  //@ assume guards[chosenService].size() > 0;
  //@ assume syncCond[chosenService];

  switch(chosenService){
    ...
    case METHOD_l:
      ...
      break;
    ...
  }
}
```

Figure 12. Server loop instrumentation for KeY when requests can be directly processed or refused. The server loop condition is changed from **`true`** to `chosenService != -1`.

For the purposes of KeY verification, it does not matter that this adaptation of the server shuts down (unlike the actual servers in Figures 5 and 9).

3.1.3. Verifying the Server Component

For each proof obligation we show how it is verified.

prop_cs_preservation: Immediately after the **switch** statement determines which branch will execute, the corresponding synchronization condition must hold. In the code generation we assert each CPRE as follows:

```
switch(chosenService){
  ...
  case METHOD_l:
    //@assert CPRE_method_l;
    ...
    break;
  ...
}
```

Since KeY cannot interpret and verify assertions, we initialize to *true* a class instance attribute, called `cprePreservation`, that will express this property. To make full correspondence with *assert* annotations, every time a service is chosen, this accumulator variable is updated (conjunctively) with the associated CPRE. At the beginning of the method this variable must be initialized as **true** and, to be verifiable in KeY we assert as an *ensure* clauses as shown in Figure 13.

```
public boolean cprePreservation;
public boolean oneMessageProcessed;
...
//@ ...
//@ ensures cprePreservation && oneMessageProcessed;
public void run(){
  ...
  cprePreservation = true;
  oneMessageProcessed = true;
  int chosenService = 42;

  while (chosenService != -1){
    int processedMessages = 0;
    chosenService = fairSelect(syncCond, services);

    switch(chosenService){
      ...
      case METHOD_l:
        cprePreservation &= CPRE_method_l;
        ...
        processedMessages ++;
        break;
      ...
    }
    oneMessageProcessed &= processedMessages == 1;
  }
}
```

Figure 13. Server code instrumentation template for verifying *prop_cs_preservation* property when when requests can be directly processed or refused.

This expresses that when a method invocation is selected, processing of the request will not violate the synchronization pre-condition for the method. Once the server knows which branch of **switch** will execute, `cprePreservation` is updated before processing the request as it is said before. The usage of `oneMessageProcessed` and `processedMessages` variables is fully described in property *prop_only_one_request*.

prop_safe_selection: services must be selected properly in each iteration. Regarding *fairness*, we rely on the proper implementation of `fairSelect`[2]. However, it is mandatory to guarantee that the selected service is valid, i.e. there is a message waiting in the channel and its associated conditional synchronization holds.

The aim of this property is to ensure that the `services` and `syncCond` arrays are well-formed and consistent to each other: (i) `services` must include all input channels and its length must be equal to $\#rg(pe)$; (ii) a channel associated with a position `i` in `services` must have its synchronization predicate in the same position of `synCond` and (iii) their lengths must be the equal. From the template, the assertions of Figure 12 are obtained.

To make them verifiable in KeY, a boolean class field, `wellFormedSyncCond`, is defined and used to check that each position of the array is consistent:

```
//@ ensures wellFormedSyncCond;
public void serverInstance(){

  wellFormedSyncCond = true;

  int[] services = {...};
  boolean[] syncCond = new boolean[#rg(pe)];
  int chosenService = 42;

  while (chosenService != -1 ) {
    ...  update syncCond array

    for (int i =0 ; i < syncCond.length ; i++ ) {
      wellFormedSyncCond &= (syncCond[i] == CPRE_i);
    }
    wellFormedSyncCond &= syncCond.length == guards.length;

    chosenService = JCSPKeY.fairSelect(syncCond, guards);
    ...  process the requests based on the chosenService
  }
}
```

This completes the instrumentation with an *ensure* clause expressing the truth of the variable.

It is important to check that every time `syncCond` is updated the resulting array is consistent. The aim of this verification is to detect when this re-evaluation is incorrect/incomplete and could allow the processing of a method invocation whose CPRE does not hold.

prop_only_one_request: This property represents that only one request is processed per server iteration. This is already guaranteed if using nested **`if`** statements, but when using **`switch`**, the execution of more than one branch is possible. The instrumentation for this property is rather simple as is shown in Figure 13; a counter variable `processedMessages` and a boolean class instance attribute, `oneMessageProcessed` are created. The first is initialized to 0 at the beginning of the server loop and it is incremented every time a requests is processed. The attribute complements the previous variable by contrasting it against 1 to express that only one request is processed at the end of each iteration. Finally, an *ensure* clause is defined to express the truth of the aforementioned attribute.

The goal of having this property is the detection of missing **`break`** statements in each **`switch`** pattern.

[2]The verification of JCSP library is out of the scope of this paper.

3.2. Deferred Requests

3.2.1. Proof Obligations

As we did for the previous template, we establish a set of proof obligations. To avoid repeating some of the explanations already given for the channel replication template that ensure that one and only one request is processed (and stored) at each iteration of the service loop, we will focus on the proof obligations that deal with the code for processing pending requests.

- *prop_cs_preservation:* immediately after the server code that retrieves a request to be processed, the CPRE of the method associated with the request must hold. Establish this CPRE, with a JML *assert*, before updating or accessing the resource – see Figure 11 (line 98 for the `get` operation and 115 for `put`). This restriction ensures *safety* of the processing code because changes to the inner state of the resources are performed only for those requests that represent valid invocations.
- *prop_completeness:* If the server exits the code for processing deferred requests – and is about to loop back to the `fairSelect` – there should be no valid pending requests. In Figure 11, (immediately after line 122), we assert that for every request in the data structure (i.e. pending), their associated CPRE does not hold.

3.2.2. Instrumentation

Instead of verifying the whole server code, we focus only on the fragment in charge of processing all deferred requests. A method encapsulating that portion of code is created and annotated with JML *ensures* clauses as before. There is no need to present any further instrumentation as there is no JCSP method or structure that impacts any variable in those JML clauses or the program flow of control.

In order to make the verification more efficient, the structure of deferred requests for each method is translated into an **`int`** variable representing its size, when CPRE and method do not depend on the input parameters. Thus, a dequeue of a requests is represented by a decrement of the given variable. Otherwise, a `List` is used as we did for the previous template.

3.2.3. Verifying the Server Component

prop_cs_preservation: The template in Figure 14 generalizes the `do-while` loop from Figure 11 (lines 88-122). Immediately after the server starts processing a deferred request, the CPRE (conditional synchronization) for the relevant operation must hold. An *assert* clause is generated to that effect.

As expressed in the previous template (Section 3.1.3), to verify that this is always honored, a new class attribute, `cprePreservation`, is initialized *true* and *and*-ed with the relevant CPRE each time an operation is processed – hopefully remaining *true* to confirm compliance. The *ensures* clause verifies whether this is the case. So long as this is positive, none of the processed requests have violated their operations' CPREs when this loop finally exits. The server then loops back to await further operation requests (as in Figure 11).

prop_completeness After the loop terminates, we need to ensure that no pending request can be processed. A request is either processed (if its CPRE holds) or enqueued again. If the former is true, the property (*prop_cs_preservation*) guarantees that is going to be processed. Otherwise, when the predicate does not hold, two cases can be distinguished. The first case, when the CPRE does not depend on the input parameters, can be described using the following *ensure* clause:

```
boolean cprePreservation;
...
//@ ensures cprePreservation;
public void processDeferredRequests(){
  cprePreservation = true;

  boolean anyResumed;
  do {
    anyResumed = false;
    ...
    // process deferred requests for operation k
    for (int i = 0; i < operation_kRequest.size()) {
      ... dequeue request item from operation_k_Request
      ... extract operation_k_footprint from the request item

      if (condition_k (operation_k_footprint) {
        /*@ assert resource_Invariant &&
                   condition_k (operation_k_footprint)
          @              ==> CPRE_k;
          @*/
        cprePreservation &= CPRE_k;  // let's see if it's true

        ... extract the channel, innerChannel, from the request item
        ... input remaining operation_k parameters, if any, from innerChannel
        ... apply operation_k to the resource, using footprint and parameters

        //@ assume resource_Invariant && POST_k;
        ... send operation_k results (or null) down innerChannel
      } else {
        ... enqueue item back on operation_k_Request
      }
    }
    ... process deferred requests for all the other operations similarly
  } while (anyResumed)
}
```

Figure 14. Server code instrumentation template for verifying *prop_cs_preservation* property when requests are deferred.

```
//prop_completeness
//@ ensures ⋀_{i=1}^{n} (methodᵢRequests > 0 ==> !CPREᵢ);
```

expressing that requests exist that cannot be processed and their associated condition does not hold. Notice that if the CPRE does not hold for a request, it will not hold for the rest of them due to the fact that the predicate only depends on the inner state of the resource.

However, when the CPRE depends on input parameters, we need to follow a similar approach as we did for the property *prop_cs_preservation*. A new variable is defined, `completeness`, and accumulates the value of the associated CPRE of requests. Finally, we conclude by *ensuring* that if there are still some requests to be processed, this variable must hold **`true`**:

```
//prop_completeness
//@ ensures ∑_{i=1}^{n} methodᵢRequest.size() > 0 ==> completeness;
```

Notice that `completeness` encapsulates all unprocessed requests and, hence, there is no need to create one variable per method.

In this case, the errors detected are related with the associated condition for treating each request. In several examples, we can present a stronger condition than the required one, leaving messages without being processed. Moreover, errors related to the *ping-pong* effect can be detected due to the fact that it also leaves processable requests enqueued.

4. Experimental Results

For a preliminar experimental study of the techniques presented, a small set of problems have been considered. These include the multibuffer that we have used as running example, some relatively small examples such as a shared resource specification of counting semaphores, a car park, and an airport traffic control tower that controls landing and taking off in several runways. We have also considered a more complex model, the *recycling plant* from [1] where steel is recovered from unsorted domestic waste with automatically controlled cranes. The growing set of examples, including specifications, documentation and (verified) code is accessible at [9]. Configuration files for reproducing the experiments in KeY are included.

For each example, we have tested both code generation techniques on correct and buggy implementations.

- Correct implementations (both approaches)
  - Implementations following the templates *in the most straightforward way*, i.e. with no *clever* optimization.
  - Optimized versions of the previous implementations. In the case of solutions based on the channel replication template, most of these optimizations are related to avoiding unnecessary recomputations of the `syncCond` array. Analogously, for deferred requests, optimizations often have to do with the data structure used to store and access the pending requests.
- Erroneous/buggy implementations
  - Channel replication:
    - Implementations with erroneous or incomplete update of the `syncCond` array.
    - Missing **`break`** statements in **`switch`** code.
  - Deferred requests:
    - Incorrect optimizations on the code processing the pending requests, e.g. leaving requests unattended, or too strong conditions on these requests.
    - Violations of *protocol* definitions.
    - Not taking into account *ping-pong* effects: processing a request of an operation B enables a request of a previously unprocessable request of operation A.

We have successfully verified in KeY all the correct implementations of the first kind and some of their optimized versions. However, on those implementations in which KeY stops the execution (some goals remain open), the tool gives back the control to the user in order to interact manually. That is, when KeY stops, it may be because the property under study is false or – a false negative – because the property holds but the theorem prover needs some assistance to establish it. In this scenario, the user can ask for a counterexample that leads to that unclosed goal – if the required prover is installed and linked, KeY will reconstruct it from the reached formula – or, alternatively, she can try to figure out whether the formula presented is valid or not, or some rule can be applied so that the proof continues.

As an example of this interactive error-finding, a version of the Readers/Writers generated using the deferred request template (Section 3.2) is analyzed[3]. The instrumented source code presented for the class `ReadersWritersCSPDeferredRequestBuggyKeY` contains two errors on lines 131 and 143 – the CPRE formula (*true*) is weaker than the expected predicate for processing the request of `afterRead` and `afterWrite`.

[3]Not shown in this paper but downloadable from [9].

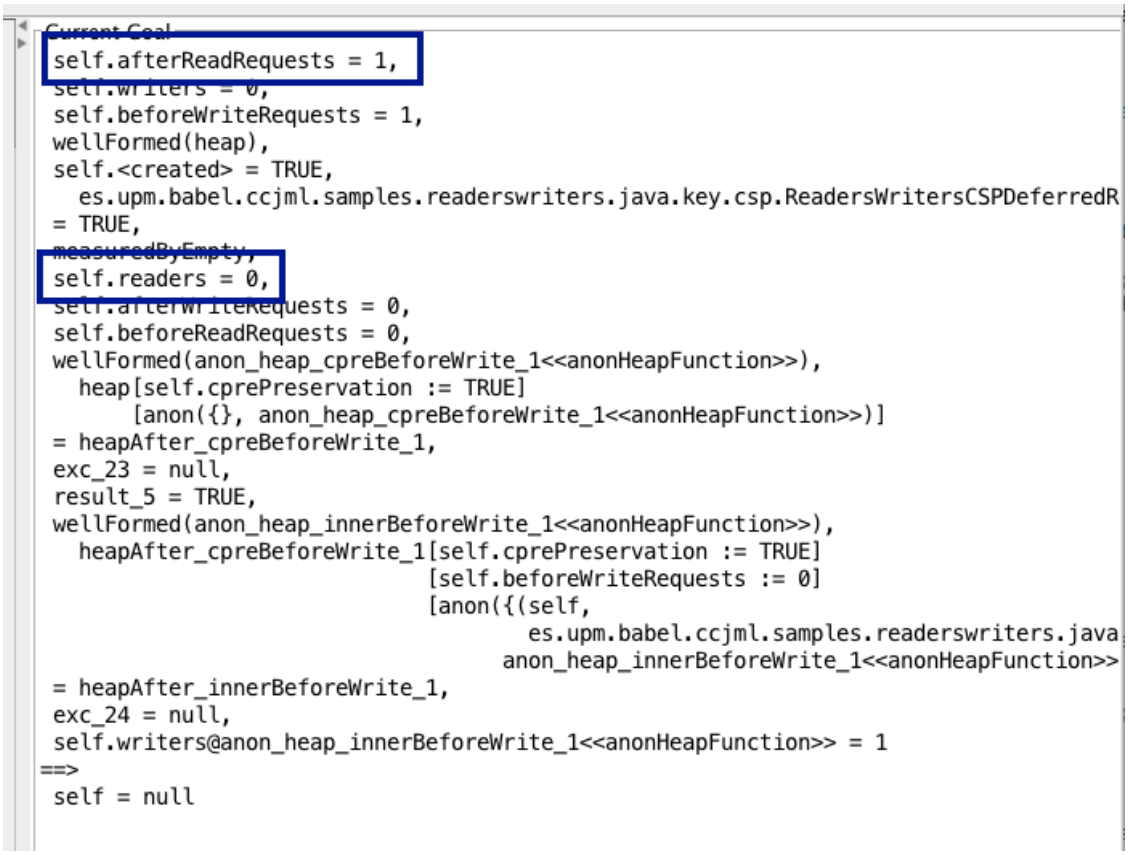

Figure 15. Unclosed goal verifying `ReadersWritersCSPDeferredRequestBuggyKeY.java`.

When trying to verify it on KeY, the tool stops with the unclosed goal shown in Figure 15. As can be seen, there are equations constraining almost all the relevant variables in the code that are rather illegible. To clarify the output, we have highlighted in boxes two pieces of the predicates, `self.readers = 0` and `self.afterReadRequests = 1`. These lines describe a scenario in which there is a request of `afterRead` (upper box) to be processed but the set of processes reading in the system is empty (lower box) leading to a clear violation of the class invariant (the `readers` variable would be decremented, taking the value `-1`).

However, the protocol specification forces that any `afterRead` invocation must be preceded by a `beforeRead` that will increment the value of `readers` attribute thus deeming this scenario impossible. Therefore, this unclosed goal can be classified as a false negative because it can never happen during correct operation of the resource by client processes.

In conclusion, the "error" detected by KeY is not caused by errors in the code but by non-enforcement of the protocol for its use. A solution to this kind of situation consists in manually strengthening some of the condition (from CPRE to CPRE + PRE) according to the protocol logic. In this case, this is enough for KeY to conclude the proof.

5. Conclusions and Future Work

We have considered the problem of generating correct Java code from high-level specifications of inter-process interaction. Given that the implementations are multi-threaded, special care must be taken to synthesize code, providing a framework that facilitates mechanical verification, as support for concurrency in Java verification tools is, for now, very scarce.

On the language side, we have extended JML with a minimal set of clauses [2] that allow shared resources to be specified as Java interfaces. This serves a twofold purpose: to make specifications closer to the realm of the programmer and to make it possible to integrate the verification process with existing tools operating on JML. The last property is the most interesting one because full verification of concurrent properties is, in general, hard to achieve.

Our approach to model-based code generation consists of semi-automatically generating partial implementations from a given shared resource specification expressed as a JML annotated interface. We have described two schemata for partial code generation based on message passing using the JCSP library. The key idea is to provide restrictive but simple templates that do not leave room to the programmer for introducing errors. Some of these can be easily detected in a purely syntactic manner, and the semantic ones are detected as violations of program assertions at specific program points obtained from the logic of the template, using the KeY tool.

Both approaches use a wrapper for the shared resource that hides message passing under the appearance of remote method invocation. The wrapper sends requests to a server thread which holds all the shared resource state. Both templates have been presented via examples, paying attention the pros and cons for applicability and how a logical analysis of the formal specification affects code generation. Code generation from the templates includes the executable code and also the JML annotations that are used proof obligations.

These annotations, completed with the instrumentation code required for KeY to reason about the JCSP constructs have allowed us to verify the adherence of the generated code to the template using a tool which does not support concurrency natively. Overall, we consider the results very promising, and we have applied these techniques to various examples available in the download site [9].

Immediate future work includes completing the experiment, i.e. fully verifying the correct implementations of the test suite, i.e. including those with "clever" but frequent optimizations. Of course, extending an existing JML compiler (such as OpenJML [12]) in order to support our interface annotations and integrating the verification method with KeY itself, thus introducing some concurrency support in it, is also on the agenda, once enough experience with the tool is gained. The instrumentation techniques used so far could serve as a basis for KeYintegration.

The practicality of more sophisticated approaches to model-driven development – e.g. *push-button* compilation from specifications – can only be assessed once humble approaches like this are fully understood.

Acknowledgements

This research has been partially founded by Comunidad de Madrid grant S2013/ICE-2731 *(N-Greens Software)* and Spanish MINECO grant TIN2012-39391-C04-03 (*StrongSoft*). We wish to acknowledge the assistance of the reviewers for their valuable comments and, with particular gratitude, to Peter H. Welch who helped us in revising and delivering this article.

References

[1] Ángel Herranz, Julio Mariño, Manuel Carro, and Juan José Moreno Navarro. Modeling concurrent systems with shared resources. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '09, pages 102–116, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] Julio Mariño, Raúl N. N. Alborodo, and Ángel Herranz. Model-based thread-safe Java code generation from JML specifications. In preparation, 2015.

[3] Ángel Herranz and Julio Mariño. A verified implementation of priority monitors in Java. In *Proceedings of the 2011 international conference on Formal Verification of Object-Oriented Software*, FoVeOOS'11, pages 160–177, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] GaryT. Leavens, AlbertL. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, volume 523 of *The Springer International Series in Engineering and Computer Science*, pages 175–188. Springer US, 1999.

[5] Peter H. Welch, Paul D. Austin, and Neil C. C. Brown. Communicating sequential processes for Java (JCSP). `http://www.cs.kent.ac.uk/projects/ofa/JCSP/`, 2012.

[6] Manuel Carro, Julio Mariño, Ángel Herranz, and Juan Jos Moreno-Navarro. Teaching how to derive correct concurrent programs (from state-based specifications and code patterns). In C.N. Dean and R.T. Boute, editors, *Teaching Formal Methods, CoLogNET/FME Symposium, TFM 2004, Ghent, Belgium*, volume 3294 of *LNCS*, pages 85–106. Springer, 2004. ISBN 3-540-23611-2.

[7] Manuel Carro, ngel Herranz, and Julio Mariño. A model-driven approach to teaching concurrency. *Trans. Comput. Educ.*, 13(1):5:1–5:19, January 2013.

[8] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.

[9] Ángel Herranz, Julio Mariño, and Raúl N. N. Alborodo. Shared resources home page. `http://babel.upm.es/~rnnalborodo/sr_web/`, 2014.

[10] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 267–276, New York, NY, USA, 2003. ACM.

[11] Juan Pablo Galeotti, Nicolás Rosner, Carlos Lopez Pombo, and Marcelo Frias. Taco: Analysis of invariants for efficient bounded verification. In Tonella and Orso, editors, *International Symposium on Software Testing and Analysis*, 2010. Trento, Italy, July.

[12] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, June 2005.