

Synthesis of Verifiable Concurrent Java Components from Formal Models

Julio Mariño ·
Raúl N. N. Alborodo ·
Lars-Åke Fredlund ·
Ángel Herranz

Received: date / Accepted: date

Abstract Concurrent systems are hard to program, and ensuring quality by means of traditional testing techniques is often very hard as errors may not show up easily and reproducing them is hard. In previous work, we have advocated a model-driven approach to the analysis and design of concurrent, safety-critical systems. However, to take full advantage of these techniques, they must be supported by code generation schemes for concrete programming languages. Ideally, this translation should be *traceable*, automated and should support the verification of the generated code. In our work, we consider the problem of generating a concurrent Java component from a high-level model of inter-process interaction (i.e. communication + synchronization). We call our formalism *shared resources*. From the model, which can be represented in mathematical notation or written as a Java interface annotated using an extension of JML, a Java component can be obtained by a semi-automatic translation. We describe how to obtain shared memory (using a priority monitors library) and message passing (using the JCSP library) implementations. Focusing on inter-process interaction for formal development is justified by several reasons, e.g.: Mathematical models are language-independent and allow to analyse certain concurrency issues, such as deadlocks or liveness properties prior to code generation. Also, the Java components produced from the shared resource model will contain all the concurrency-related language constructs, which are

often responsible for many of the errors in concurrent software. We follow a *realistic* approach where the translation is semi-automatic (schemata for code generation) and the programmer still retains the power of coding or modifying parts of the code for the resource. The code thus obtained is JML-annotated Java with proof obligations that help with code traceability and verification of safety and liveness properties. As the code thus obtained is not automatically correct, there is still the need to verify its conformance to the original specs. We illustrate the methodology by developing a concurrent control system and verifying the code obtained using the KeY program verification tool. We also show how KeY can be used to find errors resulting from a wrong use of the templates.

Keywords CSP, JCSP, KeY, Java, JML, Shared Resources, verification, model- driven, concurrency, message passing.

1 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 [17], a model-based approach was proposed as a way of alleviating some of the problems associated with developing reliable concurrent software. Some benefits of the model-based approach (both in general and, for the specific case of concurrent software) are: (i) formalizing (part of) the requirements reduces ambiguity in the problem statement; (ii) 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; (iii) code is not written from scratch but *generated* or *distilled* (semi-automatically) from the model bringing several benefits, such as *portability*, robustness against requirement changes, and the reduction of production costs; (iv) models can help in the validation, verification and test case generation of the code obtained from the previous phases.

An overview of our approach is depicted schematically in Figure 1. The core idea of our model-based approach is that the design must distinguish active entities, like clients or processes, from passive components, such as resources or shared memory locations, as shown in Figure 2. The latter represent all kinds of interaction among the former, and are modeled using an abstraction called *shared resource* that contains a clear interface – which can be invoked from pro-

J. Mariño · L-Å. Fredlund · Á. Herranz
Universidad Politécnica de Madrid
28660 Boadilla del Monte, Madrid, SPAIN
E-mail: {jmarino, lfredlund, aherranz}@fi.upm.es

R.N.N. Alborodo
IMDEA Software Institute
28223 Pozuelo de Alarcón, Madrid, SPAIN
E-mail: raul.alborodo@imdea.org

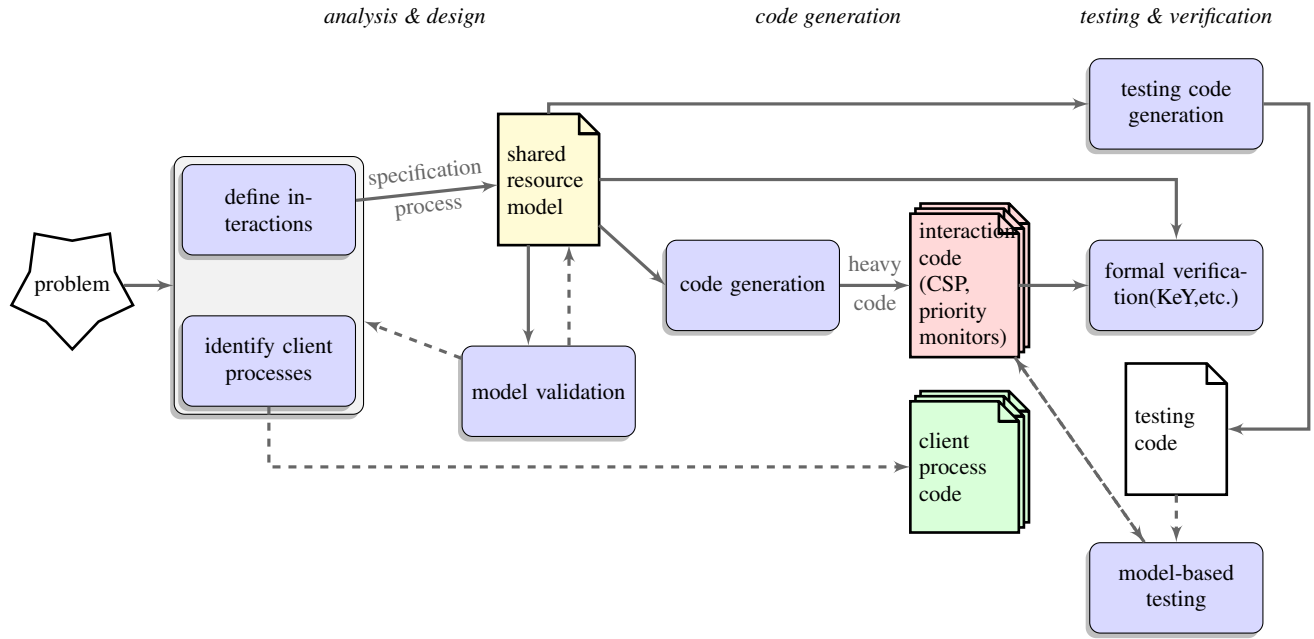


Fig. 1 Our approach to engineer verified concurrent Java software.

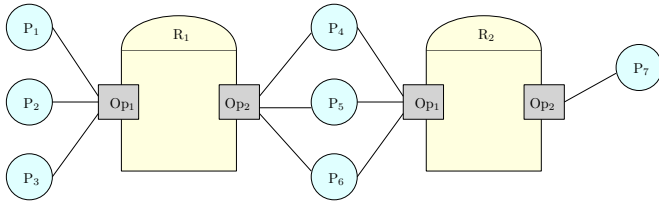


Fig. 2 System design based on shared resources: processes are shown as circles, shared resources by the larger squares (with rounded tops) and resource access methods by the smaller squares which are connected to the process invoking them.

cesses – and a transactional transition semantics. As can be seen in Fig. 1, the shared resource model is the main outcome of the analysis and design phase, as it takes part in most of the relevant activities of the software development process. Shared resource specifications are presented in Sec. 2.

The code obtained from the active entities (client process code in Figure 1) 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. In the figure, the dashed line from the client processes identified to the client processes’ code indicates that we are not considering the use of any formal activity here, i.e. the code is obtained by “traditional” means.

On the other hand, the code from the shared resources (interaction code in Figure 1) is considered *heavy* code, as it is here where all the concurrency specific constructs are 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. Code generation is the main formal activity discussed in this paper. We will describe (Sec. 3) how to translate the shared resource model into Java code using shared memory mechanisms, such as synchronized methods (Sec. 3.1), or priority monitors [16] (Sec. 3.2), and message-passing (Sec 3.3) using JCSP (*Java Communicating sequential processes* [29]) third-party library. In all cases, the structure of the code is derived from the logical structure of the interaction model, so that simple interaction models produce simple code patterns, while complex interaction models result in more intricate code.

Both the specifications and the code generated are Java files annotated using the *Java Modeling Language*(JML) [23]. For specifications, we use a minimal extension of JML devised to express shared resources as annotated Java interfaces.

We have selected JML specification language instead of others like OCL (*Object Constraint Language*¹), due to the existence of a number of JML based Java analysis tools for static checking, interactive verification, runtime checking, among others. Besides, JML specifications are just specially formatted Java comments, enabling JML annotated Java code to be understood by normal Java source code tools.

The general idea of distilling code from specifications can be approached in different ways. A *fundamentalist* approach is conceivable in which code is automatically ob-

¹ There are several patterns or tools for translating from other specification languages to JML specification sentences. An example of them is presented in [15]; in particular OCL specifications can easily be translated into JML.

tained from a specification compiler, ensuring correctness by construction and greater maintainability (through code regeneration as changes in specifications occur). For the class of concurrent programs considered in this article such an automatic translation may result in inefficient scheduling of processes, leading to unacceptable process response times. Thus, here 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.

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. One important property of our translation process is its *traceability*, i.e. the correspondence between code and specification is evident. This greatly simplifies the formal verification process, as the structure of the code is not arbitrary – as it would be if the code had been produced by traditional means – but follows a well-known pattern or template. This fact is used to define a set of proof obligations that must be checked at certain points of the code. In Sec. 4 we describe the proof obligations (for each code template) and demonstrate how these properties can be checked using an existing program verification tool (KeY [1,6]). It is important to note that KeY is a verification system for single threaded (sequential) Java code that knows nothing about concurrency. Therefore, some *tricks* are needed in order to use it as a tool for verifying concurrent code.

We conclude this overview of our formal methodology by commenting on two other formal activities not discussed in this paper. First, formal model validation is crucial to prevent certain concurrency issues before the code generation phase. Typically, properties such as liveness, absence of deadlock, etc., are much easier to analyze on a mathematical model than on actual code written in a programming language. In [17] we show one way of doing this using model checking techniques.

Second, shared resources can be used to automatise the testing process, which can be extremely complex for concurrent software. In our approach, we use the shared resource formal specification as a test oracle for deciding if a resource implementation behaves correctly or not. In [13] we discuss several ways of making this an effective way of testing implementations of shared resources.

The paper is organized as follows: the next section introduces the notation used to specify shared resources by example. Templates for code generation and their associated proof obligations follow in section 3 (each subsection is related to the concurrency mechanism used). Section 4 focuses on the experimental part where we present the code instrumentation and its verification using the KeY verification system. Section 5 gives an outline of the results obtained

during the verification phase. Finally, section 7 summarizes results, compares with related approaches and describes future work.

2 Shared Resources

The *shared resource* notation introduced in [17] expresses process interaction by defining encapsulated state, an interface of *observationally atomic* actions and a state transition semantics. We will explain here its syntax and semantics informally, by means of two running examples. A growing set of examples, including specifications, documentation and (verified) code is accessible at [2].

2.1 Readers and Writers

The *Readers & Writers* problem is a well-known example which is often used to compare and contrast different synchronization mechanisms. The task is to control concurrent reads and writes to some shared memory, to ensure that writes do not conflict with reads – or other writes.

Readers (client processes) are expected to follow a protocol in which the operation `beforeRead` is invoked first, then some shared memory is accessed, and finally `afterRead` is invoked. Writers (client processes) follow a similar protocol invoking first `beforeWrite`, performing the write operation, and then invoking `afterWrite`.

Figure 3 contains the formal specification of the synchronization pattern as a shared resource, using the syntax introduced in [17]. The resource specification defines four operations that are used to coordinate reads and writes:

- `beforeWrite`, `afterWrite` – A request for permission to write to the resource, and the notification that the Writer finished the (physical) write operation.
- `beforeRead`, `afterRead` – A request for permission to read from the resource, and the notification that the Reader finished reading the data.

The state of the resource has two fields: `readers`, the number of processes that are currently reading from the resource (a natural number), and `writers`, which counts the number of current writers.

Initially, as specified in the `INITIAL` clause, both variables are zero in order to express that there are no readers or writers. The resource has an invariant over the state, as specified by the `INVARIANT` clause, which states that writing is an exclusive activity, i.e., there can be at most one writer at any one time, and if there is a writer, there are no readers present.

The *concurrency precondition* CPRE for an operation specifies when it is safe to execute the operation – the operation is *blocked* until its CPRE becomes true. For example,

CADT ReadersWriters

OPERATIONS

ACTION BeforeRead;AfterRead;BeforeWrite;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: $writers = 1$

CPRE: *true*

AfterWrite

POST: $writers = 0$

CPRE: $writers = 0$

BeforeRead

POST: $readers = 1 + readers^{in}$

CPRE: *true*

AfterRead

POST: $readers = readers^{in} - 1$

Fig. 3 A specification of Readers & Writers resource according to the notation introduced in [17]. CADT is an acronym for *Concurrent Abstract Data Type* and represents a *shared resource*.

the CPRE for `beforeRead` permits a read request to proceed only when there is no writer present (as guaranteed by the condition $writers = 0$). The POST condition specifies the change on the resource state provoked by the completion of a call. It is possible to provide preconditions (PRE) for operations too, which specify requirements on the arguments to an operation that every call must satisfy.

An underlying assumption for the specification is that both readers and writers follow a strict protocol: an invocation of `afterRead` must be immediately preceded by an invocation of `beforeRead`, and similarly, `afterWrite` must be preceded by `beforeWrite`.

Although, for readability, we have presented the resource specification in *mathematical style*, for the purpose of Java verification all shared resources are represented as JML annotated interfaces, as introduced in [24]. 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.
- **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 become true by the changing of the shared resource inner state. In other words, this correspond 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. Other works, such as [27,3], present extensions of JML or other notations but, unlike our proposal, they rely on a specific concurrent mechanism.

We have selected JML specification language instead of others like OCL (*Object Constraint Language*²), due to the widely acceptance and support offered by several tools (including KeY Software) that cover different levels of verification such as static checking, interactive verification, runtime checking, among others. Besides, JML specifications are just specially formatted Java comments, making the source code “clean” to be handled by any Java tools.

The JML version of the readers-writers resource can be found in Figure 4.

2.2 A more complex example: *Shipping Warehouse*

A warehouse complex is serviced by a set of autonomous robots. A possible state is depicted in Fig. 5. A robot must first enter warehouse 0, then it may load an item, and next it exits warehouse 0 and enters the corridor between warehouse 0 and warehouse 1. Then, it enters warehouse 1, etc, until it finally exits the warehouse complex by exiting the last warehouse (warehouse 2 in the figure). Each robot has a weight, and the total weight of a robot and its cargo increases monotonously as it moves around in the warehouse complex. A warehouse can admit any number of robots, but to ensure safe operations the total weight of robots and their cargo cannot exceed the constant MAX_WEIGHT when a new robot enters the warehouse. It is permitted that the total weight in a warehouse is temporarily above the limit, due to loading operations, but then no more robots can be admitted to the warehouse (until a robot leaves). A corridor has place for a single robot.

² There are several patterns or tools for translating from other specification languages to JML specification sentences. An example of them is presented in [15], so OCL specifications can easily be translated into JML.

```

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();
}

```

Fig. 4 Readers & Writers specified in JML. Notice that we have added a precondition to `afterWrite` and `afterRead` operations to make the model consistent.

In Fig. 5, the constant `MAX_WEIGHT` is set to 1000kg, and thus, for example, we can see that since the total weight in warehouse 1 is $450 + 450 = 900$ the robot with weight 730 in warehouse 0 can be permitted to enter the corridor, but not warehouse 1. As the corridor between warehouse 1 and 2 is occupied, the robots inside warehouse 1 should be blocked from exiting it, until the robot occupying the corridor enters warehouse 2.

Figure 6 contains the formal specification of the control part of the robot warehouse example. The resource specification details two operations that can be used to coordinate movements between warehouses:

- `enterWarehouse(n, w)` – A request for permission for a robot to enter warehouse n carrying weight w .
- `exitWarehouse(n, w)` – A request for permission for a robot to exit a warehouse n towards a corridor carrying weight w .

The state of the resource has two fields: `weight`, a mapping from a warehouse to weight (a natural number), and `occupied`, a mapping from a warehouse to a boolean. Intu-

itively, `weight` should correspond to the accumulated weight in the warehouse, and `occupied[n]` is true if there is a robot present in the corridor n leading from the warehouse.

Initially, the weight in all warehouses is zero, and no robot is present in any corridor. The `INVARIANT` clause describes that the weight in a warehouse should always be less than or equal to the maximum weight `MAX_WEIGHT`.

A robot that wants to enter warehouse n with weight w should first call `enterWarehouse(n, w)` to ask the resource (controller) for permission to do so. It is the task of the (implemented) resource to ensure that the call does not return (i.e., that it *blocks*) until it is safe for the robot to enter the warehouse. The CPRE specifies that the access is safe when the accumulated weight of the robots already in the warehouse plus the weight of the new robot is less than or equal to the allowed maximum weight. The `POST` condition specifies the change on the resource state provoked by the completion of a call. It is possible to provide preconditions (PRE) for operations too, which specify requirements on the arguments to an operation that every call must satisfy.

Similarly, a robot should always call the operation `exitWarehouse(n, p)` to ask for permission to leave a warehouse. The CPRE condition of the resource specification ensures that the call does not return until the corridor leading away from the warehouse is free from robots. A restriction on the caller to these operations is that the weight w provided as argument to the operation `exitWarehouse(n, w)` when asking for permission for leaving a warehouse, must be identical to the weight provided when asking for permission to enter the warehouse, i.e., `enterWarehouse(n, w)`. That is, the exit weight should not reflect any cargo loaded in the warehouse, instead, the weight increase should be factored into the next call to `enterWarehouse`, e.g., `enterWarehouse(n+1, w+cw)`.

Note that the control system does not need to identify individual robots, and indeed cannot, as there is no robot identifier passed along in the two operations. Moreover note that the example abstracts away from the use of e.g. scales to weigh robots and cargo, i.e., it is assumed that calls to these two operations always identify the correct weights of robots.

As in the previous example, the Java interface containing the specification for the shipping warehouse is shown in Figure 7.

3 Model-based Code Generation for Shared Resources

The following subsections present an approach for translating a shared resource specification into a Java class implementation, using a number of code templates. Such a template specifies the code refactoring needed to implement a shared resource, as well as, crucially, specifying how the

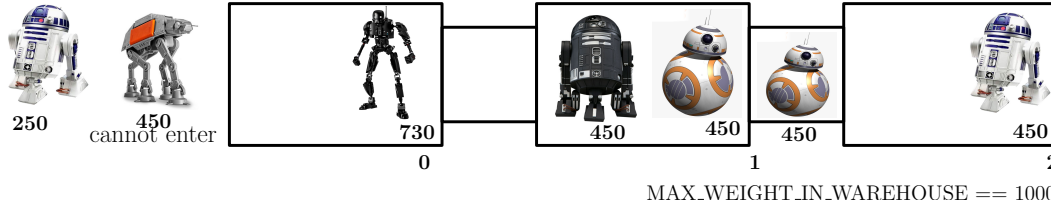


Fig. 5 Warehouses and robot movements.

CADT WarehouseAccessControl

OPERATIONS

ACTION enterWarehouse: $\mathbb{N}[i] \times \mathbb{N}[i]$ ACTION exitWarehouse: $\mathbb{N}[i] \times \mathbb{N}[i]$

SEMANTICS

DOMAIN:

STATE: ($weight : Warehouse \rightarrow Weight \times$
 $occupied : Warehouse \rightarrow \mathbb{B}$)TYPE: $Warehouse = 0 \dots N_WAREHOUSES - 1$
 $Weight = 0 \dots MAX_WEIGHT$ INITIAL: $\forall n \in Warehouse \bullet weight(n) = 0 \wedge \neg occupied(n)$ INVARIANT: $\forall n \in Warehouse \bullet weight(n) \leq MAX_WEIGHT$ PRE: $n \in \{0 \dots N_WAREHOUSES - 1\} \wedge$
 $w \in \{0 \dots MAX_WEIGHT - 1\}$ CPRE: $w + weight(n) \leq MAX_WEIGHT$

enterWarehouse(n,w)

POST: $weight = weight^{in} \oplus \{n \mapsto weight^{in}(n) + w\} \wedge$
 $(n > 0 \Rightarrow occupied = occupied^{in} \oplus \{n \mapsto \text{False}\}) \wedge$
 $(n = 0 \Rightarrow occupied = occupied^{in})$ PRE: $n \in \{0 \dots N_WAREHOUSES - 1\} \wedge$
 $w \in \{0 \dots MAX_WEIGHT - 1\}$ CPRE: $n = N_WAREHOUSES - 1 \vee \neg occupied(n)$

exitWarehouse(n,w)

POST: $weight = weight^{in} \oplus \{n \mapsto weight^{in}(n) - w\} \wedge$
 $(n < N_WAREHOUSES - 1 \Rightarrow$
 $occupied = occupied^{in} \oplus \{n \mapsto \text{True}\}) \wedge$
 $(n = N_WAREHOUSES - 1 \Rightarrow occupied = occupied^{in})$

Fig. 6 Shipping Warehouse specification according to the notation introduced in [17].

JML code annotations of the shared resource interface are transformed into JML code annotations of the implemented shared resource.

Selecting a suitable code template is a two-step process. First, a particular concurrency mechanism is selected, e.g., using Java synchronized methods, priority monitors, or using the JCSP message-passing library. The choice of which concurrency mechanism to use is largely a matter of taste, although it should be noted that the single code template we present for Java synchronized methods cannot be recommended in general, as it does not guarantee important (process) liveness properties. Second, having chosen a concurrency mechanism, a template user should analyse the concurrency preconditions (CPRE) and post conditions – e.g.,

```
public interface /*@ shared_resource @*/
    WarehouseAccessControl {
    /*@ public model instance int N_WAREHOUSES;
    /*@ public model instance int MAX_WEIGHT;

    // corridor N is between warehouse N and N-1
    /*@ public model instance boolean[] occupied;
    /*@ public model instance int[] weight;

    /*@ public initially
    @ (\forallall int i; i >= 0 && i < N_WAREHOUSES;
    @     weight[i] == 0) &&
    @ (\forallall int i; i >= 0 && i < N_WAREHOUSES;
    @     !occupied[i]);
    @*/

    /*@ public instance invariant MAX_WEIGHT > 0 &&
    @     occupied.length == N_WAREHOUSES &&
    @     weight.length == N_WAREHOUSES &&
    @     (\forallall int i; i >= 0 && i < N_WAREHOUSES;
    @         weight[i] <= MAX_WEIGHT);
    @*/

    /*@ public normal_behaviour
    @ requires n => 0 && n < N_WAREHOUSES &&
    @     w > 0 && w < MAX_WEIGHT;
    @ cond_sync weight[n] + w <= MAX_WEIGHT;
    @ assignable weight[n], occupied[n-1];
    @ ensures weight[n] == w + \old(weight)[n] &&
    @     (n > 0 ==> !occupied[n-1]);
    @*/
    public void enterWarehouse(int n, int w);

    /*@ public normal_behaviour
    @ requires n => 0 && n < N_WAREHOUSES &&
    @     w > 0 && w < MAX_WEIGHT;
    @ cond_sync n = N_WAREHOUSES - 1 || !occupied[n];
    @ assignable weight[n], occupied[n];
    @ ensures weight[n] == \old(weight)[n] - w &&
    @     (n < N_WAREHOUSES ==> occupied[n]);
    @*/
    public void exitWarehouse(int n, int w);
}
```

Fig. 7 The Warehouse example specified in JML

to find out if the concurrency precondition depends on both the operation parameters and the resource attributes – i.e. its internal state. Next, a particular code template is selected which is compatible with the analysis result.

In the following subsection we thus define a number of code templates: Sect. 3.1 presents a single code template for implementing a shared resource using Java synchronized methods, while Sect. 3.2 presents a number of code templates for implementation using priority monitors. Finally, Sect. 3.3 contains code templates for implementing resources using the JCSP Java CSP messaging library. These code templates originate from the classroom use of the model-

driven approach in several undergraduate courses on concurrency [11, 10].

3.1 Model-based Code Generation Using Synchronized Methods

The first template takes advantage of Java's *synchronized* method declaration to guarantee mutual exclusion. This synchronization mechanism uses the locks native to the Java Object class to prevent two or more threads to access simultaneously the code in these methods marked with the *synchronized* modifier. In addition to mutual exclusion, a very basic mechanism for condition synchronization is provided. A thread inside a synchronized method can suspend by invoking *wait()*, which also releases the mutual exclusion lock, allowing another thread to enter the code in some synchronized method of that object. On the other hand, threads that modify the state of the shared object may invoke *notifyAll()*, which gives the suspended threads a chance to resume execution (when the mutual exclusion lock permits).

The generic template for translating an operation *op* of the given shared resource, with parameters $arg_1, \dots, arg_M$, to Java code using synchronized methods is presented in Figure 8.

This template, as well as the ones presented in the following sections, is a combination of immutable parts, and placeholders (marked as gray boxes) for fragments of Java code derived from the specification of the shared resource being implemented. Currently these Java fragments must be filled in manually by the user – an item for future work is to enable such code fragments to be derived automatically from a shared resource specification.

Given an operation *op* and its associated mathematical predicates PRE, CPRE and POST representing the pre-, concurrency pre- and post-condition as shown in the shared resource specification (Figures 3 and 6 for Readers-Writers and the Shipping Warehouse problems, respectively), the following total functions are defined:

$$Pred2JML : predicate \rightarrow JML_expression$$

which maps every predicate into a JML expression;

$$Pred2Java : predicate \rightarrow Java_boolean_block$$

which takes a mathematical predicate from the shared resource specification and converts it into a Java boolean expression³;

$$Post2Java : predicate \rightarrow Java_block$$

³ For simplicity, we are assuming that the Java code for checking a boolean condition can be represented by a call to an auxiliary method, which would be part of the outcome of *Pred2Java*. Of course, actual code may take a different, but equivalent, form.

```

1 public synchronized TN op1(T1 arg1, ..., TM argM)
2     throws PreViolationResourceException{
3
4     if (!Pred2Java(PREop1(a1, ..., an))) {
5         throw new PreViolationResourceException();
6     }
7     /*@ assume Pred2JML(PREop1(a1, ..., an));
8     TN result = null;
9     while (!Pred2Java(CPREop1(a1, ..., an))) {
10         wait();
11     }
12     /*@ assert Pred2JML(PREop1(a1, ..., an))
13     @    && \invariant_for(this)
14     @    && Pred2JML(CPREop1(a1, ..., an));
15     @*/
16     Post2Java(POSTop1(a1, ..., an));
17     /*@ assert Pred2JML(POSTop1(a1, ..., an))
18     @    && \invariant_for(this);
19     @*/
20     notifyAll();
21     return result;
22 }

```

Fig. 8 Generic template for implementing an operation of a shared resource using synchronized methods.

converts a post-condition predicate into a block of code that establishes the postcondition.

For instance, let us consider *enterWarehouse* operation of the Shipping Warehouse problem specified in Figure 6. In this case, *Pred2Java* and *Pred2JML* return the same expression for both PRE and CPRE. The Java boolean expression for the precondition is shown below:

$$\begin{aligned}
 Pred2JML(pre_{enterWarehouse(n,w)}) = \\
 (n \Rightarrow 0 \ \&\& \ n < N_WAREHOUSES \ \&\& \\
 w > 0 \ \&\& \ w < MAX_WEIGHT)
 \end{aligned}$$

Similarly, a Java code fragment that encodes the *POST* predicate is shown below:

$$\begin{aligned}
 Post2Java(post_{enterWarehouse(n,w)}) = \\
 \{ \\
 \quad \text{if } (n \neq 0) \\
 \quad \quad occupied[w-1] = false; \\
 \quad \quad weight[n] += w; \\
 \}
 \end{aligned}$$

The template first checks the precondition (PRE_{op}) of the given operation, and raises the an instance of the exception *PreViolationResourceException* if not satisfied. The arguments $a_1, \dots, a_n$ to PRE_{op} correspond to the (some) operation arguments ($arg_1, \dots, arg_M$) and the resource attributes. After the precondition has been checked, a JML annotation is inserted, an assumption that the precondition holds.

Next the concurrency precondition (CPRE_{op}) is repeatedly checked, until it holds. A call to *wait* is inserted to permit the calling thread to sleep until the resource state changes. The next line in the code templates provides a JML annotation which states that the both the precondition, the

concurrent precondition and the resource invariant are asserted.⁴ Next, the code obtained from *Post2Java* for the method is executed, i.e., the resource attributes are updated, and the operation result is also computed. In the next line a JML annotation asserts the post condition ($POST_{op}$) and the invariant of the shared resource. Finally, the call to `notifyAll()` permits another waiting operation to proceed, and the operation result is returned.

The result of applying this code template to all the operations of a shared resource is a set of Java methods, which should be embedded in a single Java class containing the attributes of the resource represented as Java attributes.

Note that the process of applying a code template to a object method to obtain a Java method is still a manual activity. As future work, we intend to implement code templates as code refactoring activities in a Java integrated development environment.

ReadersWriters Implementation (using synchronized methods) By applying the code template in Figure 4 to the Readers & Writers shared resource in Figure 8 we can derive the implementation in Figure 9.

Observe that the code generated from this template trivially satisfies the serializability criterion for shared resources. As the only statements that can affect the state of the object are assumed to be in the body of the operation (for instance, line 39 of `beforeRead` operation), and no suspension occurs inside it, all possible interleaving of method invocations are equivalent to a sequence of disjoint executions of core/body statements.

Considering the operation `beforeRead` of Figure 9, the *precondition* (PRE) must hold at call time (line 34). Then, immediately before performing the modifications in the resource, it is mandatory to check if its *synchronization condition* (CPRE - line 35) holds. If it does not hold, the invoker is blocked inside the object's lock (line 35). However, when its CPRE holds, the core statement (body of the method - line 39) updates the resource state according to the *postcondition* (expressed as an *ensures* JML clause - ls. 40-42). Finally, the method calls `notifyAll` in order to wake up all threads that are waiting on this object's lock.

Most of the conditions above follow trivially from the way the template is designed. For example, considering the `beforeRead` operation and assuming the implementation of CPRE is pure and correct, the loop itself ensures that CPRE

⁴ Notice that JML *assume* statements are those expressions on which the code can 'trust' to hold. In the case of the PRE, it is assumed that the expression is valid before continuing the execution. However, *assert* expressions must be checked and if some of those expressions do not hold, the code must halt or warn the user that the system is in an inconsistent state. In the case of the CPRE, the expression must hold before continuing executing the critical section of the operation indicating that the invariant holds and the access to the shared resource inner state variables is allowed.

```

1 public class ReadersWritersSync {
2
3     protected int readers = 0;
4     protected int writers = 0;
5
6     public synchronized void beforeWrite() {
7         //@ assume true;
8         while (!(readers == 0 && writers == 0){
9             wait();
10        }
11        /*@ assert true && \invariant_for(this)
12           @      && readers == 0 && writers == 0;
13           @*/
14        ++writers;
15        /*@ assert writers == 1 && \invariant_for(this);
16           @      && true;
17        */
18        notifyAll();
19    }
20
21    public synchronized void afterWrite()
22        throws PreViolationResourceException{
23        if (!(writers == 1))
24            throw new PreViolationResourceException();
25        /*@ assume writers == 1;
26           @      && true;
27           @*/
28        while (!true) { wait(); }
29        /*@ assert writers == 1 && \invariant_for(this)
30           @      && true;
31           @*/
32        --writers;
33        /*@ assert writers == 0 && \invariant_for(this);
34           @      && true;
35        */
36        notifyAll();
37    }
38
39    public synchronized void beforeRead() {
40        //@ assume true
41        while (!writers == 0){ wait(); }
42        /*@ assert true && \invariant_for(this) &&
43           @      writers == 0;
44           @*/
45        ++readers;
46        /*@ assert readers == \old(readers) + 1
47           @      && \invariant_for(this);
48           @*/
49        notifyAll();
50    }
51
52    public synchronized void afterRead()
53        throws PreViolationResourceException{
54        if (!(readers > 0))
55            throw new PreViolationResourceException();
56        /*@ assume readers > 0;
57           @      && true;
58           @*/
59        while (!true) { wait(); }
60        /*@ assert readers > 0 && \invariant_for(this)
61           @      && true;
62           @*/
63        --readers;
64        /*@ assert readers == \old(readers) - 1
65           @      && \invariant_for(this);
66           @*/
67        notifyAll();
68    }
69 }

```

Fig. 9 Readers & Writers implementation using synchronized methods from the interface in Figure 4. The invariant is omitted in the specifications to make the specification easier to read.

holds right before `++readers`. Most of the work of mechanically verifying these properties would go in proving

```

{true ∧ Invariant ∧ writers == 0}
++readers;
{readers == \ old(readers) + 1 ∧ Invariant}.

```

However, although the solution presented is valid in the sense that the safety conditions are met, repeatedly checking the synchronization condition in a loop makes it quite inef-

ficient in the presence of a large number of threads. Moreover, the nondeterminism inherent to `notifyAll()` makes it impossible to put an upper bound to the number of times threads execute the condition synchronization loops at the entry of `put` and `get` – i.e. liveness properties such as *bounded wait* cannot be ensured with this scheme.

3.2 Model-based Code Generation Using Priority Monitors

The problems with the `wait()/notifyAll()` mechanism are caused by its inability to force the resumption of threads whose synchronization condition is known to hold. *Monitors* [7, 19] were designed to allow for a finer control of thread blocking and resumption. Although Java includes its own realization of the monitor concept in the *locks & conditions* library, we will present a quite efficient implementation scheme based on our own implementation of *priority monitors* [16].⁵

Our monitor library provides a class `Monitor` of monitor objects which, in first place, can be used to enforce mutual exclusion on critical code sections. A monitor object (say `mutex`) acts as a lock which is grabbed by executing `mutex.enter()` and then released by executing `mutex.leave()`. Also, as a way to specify condition synchronization, a monitor object can have a set of associated *condition queues*⁶ of class `Monitor.Cond`. These can be used as dedicated wait queues for threads with specific synchronization needs. A thread willing to block will execute `q.await()` – that will release the lock on the monitor to which `q` is associated – and threads can mark blocked threads as ready to resume by invoking `q.signal()`. The “priority” tag comes from the fact that when some thread executes `q.signal()` on a nonempty queue right before executing `mutex.leave()` the signaled thread is guaranteed to resume execution *right after the signaling one*, i.e. there is no possibility of a third thread entering the monitor in between. As we will see shortly, this property is crucial for the scheme we are proposing. In order to ensuring that signals are effective, invoking `q.waiting()` returns the number of threads blocked on `q`. Of course, the `await()`, `signal()` and `waiting()` methods can only be invoked by a thread holding the monitor associated with the condition.

The key point for the scheme to work properly is to decide how many condition queues are required to block threads and how to manage them for signaling. The basic requirement is that every condition q must have an associated assertion W_q that is proven to hold every time `q.signal()` is called and such that W_q logically implies the CPRE for ev-

ery thread blocked on q .⁷ In our model-based approach this decision is made after a logical analysis of the CPRES. We will consider four cases:

I) The CPRE is *true*. In this case, no condition queue is required and no blocking will take place prior to entering the operation code.

II) The CPRE is a formula different from *true* but independent from the operation’s inputs (it might depend, however, on the resource’s internal state). In this case we consider a single condition queue for this operation as blocking or awakening a thread will only depend on the resource state.⁸

In general, such a simple scheme is not applicable. 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. When CPRES depend on the value of one or more of their parameters a one-to-one correspondence between operations and condition queues is no longer possible. However, variations on the previous scheme are still possible that preserve the 0-1 principle. Because of that and taking into account the following definition, two more schemata are presented.

Definition 1 Consider one operation op that takes arguments x and $\bar{y}$. Let D_x, D_y be the domains of $x, \bar{y}$ and C the CPRE for op . C is independent from y iff

$$\forall a \in D_x. \forall b, b' \in D_y. C_i[a/x, b/y] \Leftrightarrow C_i[a/x, b'/y]$$

C depends on x iff $\exists a, a' \in D_x. C_i[a/x] \not\equiv C_i[a'/x]$. C partitions D_x into a set of equivalence classes trivially: $a, a' \in D_x$ are equivalent iff $C[a/x]$ and $C[a'/x]$ are logically equivalent.

III) [PARAMETER INDEXATION] The CPRE is a formula that depends on some input for the operation. The set E of equivalence classes induced by that input is finite – and “small”, we should say. Then a condition for each equivalence class is created. If the CPRE does not hold at call time, then the call is blocked on the condition queue corresponding to its equivalence class.

However, when the set of equivalence classes is infinite – or too large – the solution above cannot be applied. One possibility is to assume that the number of active threads is finite, and then each thread can be blocked in a dedicated condition queue. We call this scheme *client indexation*:

IV) [CLIENT INDEXATION] When a thread calls an operation and the CPRE does not hold, a condition queue is

⁷ This idea can be traced back to [19].

⁵ A survey on the different monitor flavours, with an analysis of the pros and cons of each proposal can be found in [8].

⁶ Also known as “condition variables”, “event queues”, etc., in the literature.

⁸ Actually, we are assuming that the only distinction between threads blocked on such an operation is calling time and thus an FCFS policy is fine. Of course, this might not always be the case and then input parameters should be taken into account. For simplicity, we are not considering this possibility now.

```

1 private Monitor mutex;
2 ...
3 private Monitor.Cond condOpi;
4 ...
5
6 public TN opi(T1 arg1, ..., TM argM)
7     throws PreViolationResourceException{
8
9     mutex.enter();
10    if (!Pred2Java(PREopi(a1, ..., an))) {
11        mutex.leave();
12        throw new PreViolationResourceException();
13    }
14    /*@ assume Pred2JML(PREopi(a1, ..., an));
15    TN result = null;
16    if (!Pred2Java(CPREopi(a1, ..., an)))
17        condOpi.await();
18
19    /*@ assert Pred2JML(PREopi(a1, ..., an))
20    @      && \invariant_for(this)
21    @      && Pred2JML(CPREopi(a1, ..., an));
22    @*/
23    Post2Java(POSTopi(a1, ..., an));
24    /*@ assert Pred2JML(POSTopi(a1, ..., an))
25    @      && \invariant_for(this);
26    @*/
27
28    // signaling code
29    int signaled = 0;
30    for each condOpj && signaled == 0{
31        if (condOpj.waiting() > 0) {
32            condOpj.signal();
33            /*@ assert Pred2JML(CPREopi());
34            signaled++;
35            break;
36        }
37    }
38    /*@ assert signaled >= 0 && signaled <= 1;
39    /*@ assert Pred2JML(POSTopi(a1, ..., an))
40    @      && \invariant_for(this);
41    @*/
42    mutex.leave();
43    return result;
44 }

```

Fig. 10 Generic template for implementing a shared resource using priority monitors when CPRE does not depend on any parameter.

created a new and stored in some data structure along with the inputs on which it depends. After that, the thread blocks on the condition. Right after waking up the condition queue and the associated input data can be deleted.

3.2.1 Trivial Cases

Figure 10 shows a generic template for code generation using priority monitors. The code for all the operations follows a similar pattern. First, mutual exclusion is ensured by enclosing the whole operation in a monitor `enter()`/`leave()` pair. Then the rest of the code is structured in three blocks. The first one evaluates the CPRE and blocks the calling thread – by waiting on some condition queue – if it does not hold. The second code block (ln. 23) corresponds to the operation itself and must establish the POST. Finally, the third block chooses some blocked thread that can be resumed, by

executing a `signal()` right before leaving the monitor and returning. In the example shown the four operations share the unblocking code but this need not always be the case.

It is worth noting that the only point where sequential execution can be interrupted – if any – is the `await()` in the first segment. After that, the CPRE is assumed to hold, either because the CPRE was true at call time and the `await()` was not invoked or, because the `await()` was invoked once and some other thread invoked `signal()` after checking that the CPRE for the blocked thread was satisfied after completing operation. That is, blocked threads do not need to reevaluate their CPRES on resumption, but this requires them to check some other thread’s CPRE on exit. We call this the *O/I scheme*, i.e. at most one `await()` is executed by a thread during the monitor call and at most one `signal()`.⁹

Readers & Writers Implementation (using priority monitors)

The first two cases are rather trivial but cover a small class of interesting resources, including the readers & writers protocol which code is shown in Figure 11. In this example, operations *afterRead* and *afterWrite* fall in case (I) and operations *beforeRead* and *beforeWrite* in case (II). See [2] for details on other examples of this kind.

3.2.2 Parameter Indexation

When the CPRE depends on at least one of the method parameters and set of equivalence classes induced is “small”, then a given call is blocked on the condition corresponding to the associated equivalence class.

Generic Template A generic template for this approach is presented in Figure 12. Instead of having one condition queue for each operation, we create a set of conditions: one for each of the equivalence classes representatives defined over the domain of the dependent parameters, as is expressed in lines 2-4. Then, if the CPRE does not hold, the call is blocked on the condition associated with the representative of the equivalence class of the current value of the parameter, as it is shown in line 16-17.

Then, the programmer is in charge of writing the code that must accomplish the POST of the method.

Following with the unblocking part, the code has to ensure that the associated predicated of the signaled condition queue implies the CPRE of the method call, assuring that the awoken thread can safely continue with the execution.

Shipping Warehouse Implementation (by Parameter Indexation)

Let us illustrate the parameter indexation method with

⁹ The library allows to issue more than one signal before releasing the monitor but then it can get rather difficult to ensure that CPRES hold on resumption, so the template presented does not consider that possibility.

```

1 public class ReadersWritersMonitorNaive {
2   protected int readers = 0, writers = 0;
3
4   protected Monitor mutex = new Monitor();
5   protected Cond writersCond = mutex.newCond();
6   protected Cond readersCond = mutex.newCond();
7
8   public void beforeRead() {
9     mutex.enter();
10    if (! writers == 0)
11      readersCond.await();
12    ++readers;
13
14    dummyUnblockingCode();
15    mutex.leave();
16  }
17
18  public void afterRead() {
19    mutex.enter();
20    if (!true) readersCond.await();
21    --readers;
22
23    dummyUnblockingCode();
24    mutex.leave();
25  }
26
27  public void beforeWrite() {
28    //@ assume true;
29    mutex.enter();
30    if (!(readers == 0 && writers == 0))
31      writersCond.await();
32
33    /*@ assert true && \invariant_for(this)
34     @      && readers == 0 && writers == 0;
35     @*/
36    writers++;
37    //@ assert writers==1 && \invariant_for(this);
38
39    dummyUnblockingCode();
40    //@ assert writers==1 && \invariant_for(this);
41    mutex.leave();
42  }
43
44  public void afterWrite() {
45    //@ assume writers = 1 ;
46    mutex.enter();
47    if (!true) writersCond.await();
48
49    /*@ assert writers==1 && \invariant_for(this)
50     @      && true;
51     @*/
52    writers--;
53    //@ assert writers==0 && \invariant_for(this);
54
55    dummyUnblockingCode();
56    //@ assert writers==0 && \invariant_for(this);
57    mutex.leave();
58  }
59
60  protected void dummyUnblockingCode() {
61    int signaled = 0;
62    if (readersCond.waiting() > 0 && writers == 0){
63      signaled++;
64      readersCond.signal();
65    } else if (writersCond.waiting() > 0 &&
66      writers == 0 && readers == 0){
67      signaled++;
68      writersCond.signal();
69    }
70    //@ assert signaled >= 0 && signaled <= 1;
71  }
72 }

```

Fig. 11 Readers & Writers implementation using priority monitors with unoptimized unblocking code. JML annotations were only added to *write* operations (*beforeWrite()* and *afterWrite()*). PRE verification code is also omitted.

```

1 private Monitor mutex;
2 private Monitor.Cond q1;
3 ...
4 private Monitor.Cond qM;
5
6 public TN opl(T1 arg1, ..., TM argM)
7   throws PreViolationResourceException{
8
9   mutex.enter();
10  if (!Pred2Java(PREopl(a1,...,an))) {
11    mutex.leave();
12    throw new PreViolationResourceException();
13  }
14  //@ assume Pred2JML(PREopl(a1,...,an));
15  TN result = null;
16  if (!Pred2Java(CPREopl(a1,...,an)))
17    qj.await();
18
19  /*@ assert Pred2JML(PREopl(a1,...,an))
20   @      && \invariant_for(this)
21   @      && Pred2JML(CPREopl(a1,...,an));
22   @*/
23  Post2Java(POSTopl(a1,...,an));
24  /*@ assert Pred2JML(POSTopl(a1,...,an))
25   @      && \invariant_for(this);
26   @*/
27
28  // signaling code
29  int signaled = 0;
30  for each qi && signaled == 0{
31    if (qi.waiting() > 0 && pred_qi) {
32      qi.signal();
33      //@ assert pred_qi ==> Pred2JML(CPREopi());
34      signaled++;
35      break;
36    }
37  }
38  //@ assert signaled >= 0 && signaled <= 1;
39  /*@ assert Pred2JML(POSTopl(a1,...,an))
40   @      && \invariant_for(this);
41   @*/
42  mutex.leave();
43  return result;
44 }

```

Fig. 12 Generic template for implementing a shared resource using the parameter indexation technique.

the shipping warehouse code shown in Fig. 13 . The CPRE of *enterWarehouse* depends on the *n* and *w* parameters. As $D_n = 1..N_WAREHOUSES$ and $D_w = 1..MAX_WEIGHT$, the set E_{ew} of equivalence classes is isomorphic to $D_n \times D_w$ and thus a bidimensional array *freedWarehouse* of condition queues is used. For *exitWarehouse*, however, the set E_{xw} of equivalence classes is isomorphic to D_n and a two-dimension array *freedCorridor* is used instead. By construction, the condition associated to each queue *freedCorridor*[*i*][*j*] is exactly $CPRE(\text{enterWarehouse}(n, w))[i/n, j/w]$ and thus the basic requirement is trivially fulfilled.

In our example, having one condition per warehouse *and* weight can be considered an overkill. client indexation can be used to optimize the previous schema in several ways. First, if we consider the robots trying to enter warehouse

```

1 public class WarehouseAccessControlMonitorNaive {
2   // Resource state
3   private int weight[];
4   private boolean occupied[];
5
6   // Handling concurrency
7   // ensures mutual exclusion
8   private Monitor mutex;
9   private Monitor.Cond freedWarehouse[][] =
10     new Monitor.Cond[N_WAREHOUSES][MAX_WEIGHT];
11   private Monitor.Cond freedCorridor[] =
12     new Monitor.Cond[N_WAREHOUSES];
13
14   public WarehouseResource() {
15     // initialize state monitors and conditions...
16   }
17
18   public void enterWarehouse(int n, int w) {
19     mutex.enter();
20     // Check CPRE coded in Java (just once)
21     if (w + weight[n] > MAX_WEIGHT)
22       freedWarehouse[n][w].await();
23     //@ assert weight[n] + w <= MAX_WEIGHT;
24
25     if (n != 0) occupied[w-1] = false;
26     weight[n] += w;
27
28     // look for some thread to be signaled
29     // and release the monitor
30     dummyUnblockingCode();
31     mutex.leave();
32   }
33
34   public void exitWarehouse(int n, int w) {
35     mutex.enter();
36     // Check CPRE coded in Java (just once)
37     if (n < N_WAREHOUSES - 1 && occupied[n+1])
38       freedCorridor[n+1].await();
39     //@ assert n == N_WAREHOUSES-1 || !occupied[n];
40
41     weight[n] -= w;
42     if (n != N_WAREHOUSES - 1) occupied[n] = true;
43
44     // look for some thread to be signaled
45     // and release the monitor
46     dummyUnblockingCode();
47     mutex.leave();
48   }
49
50   private void dummyUnblockingCode(){
51     int signaled = 0;
52     // try to wake up a robot trying to enter
53     // for each warehouse...
54     for(int i=0; i<N_WAREHOUSES && signaled==0; i++){
55       // and for each possible weight...
56       for (int j=0; j <= MAX_WEIGHT && !signaled; j++) {
57         // we seek for blocked robot
58         if (weight[i] + j <= MAX_WEIGHT //cpreEnter
59           && freedWarehouse[i][j].waiting() > 0){
60           freedWarehouse[i][j].signal();
61           signaled ++;
62         }
63       }
64     }
65
66     // trying to wake up a robot trying to exit
67     // for each warehouse...
68     for(int i=0; i<N_WAREHOUSES && signaled==0; i++){
69       // we seek for a blocked robot
70       // that could exit a warehouse
71       if ((i==N_WAREHOUSES-1 || occupied[i]) &&
72         freedCorridor[i].waiting() > 0){
73         freedCorridor[i].signal();
74         signaled ++;
75       }
76     }
77   }
78   //@ assert signaled >= 0 && signaled <= 1;
79 }
80 }

```

Fig. 13 An implementation sketch of the Warehouse resource Figure 7. The invariant is omitted in the specifications to make the specification easier to read.

zero, rather than having an array of conditions indexed by weight, we can just store pairs $\langle q_i, w_i \rangle$. Moreover, if a *priority queue* is used, several strategies can be implemented, such as *lightest robot first*, etc. Second, for the rest of warehouses a more radical optimization can be applied. As there is only room for one robot in the corridor, we can assume that at most one thread is blocked trying to enter warehouse $i > 0$, so just one queue and one weight will suffice. The code for these optimized versions can be found in [2].

In the version shown in Fig. 13 the unblocking code is shared among the two operations. It just iterates over the condition arrays of threads and checks: (1) that the assertion for a given queue holds, and (2) that there is a thread actually blocked on that queue (using the `waiting()` method). If so, the thread is awakened and the iteration stops to preserve the 0/1 scheme.

Ensuring (1) requires that the signaling thread is able to recover the safety assertion W_q for a given condition q . In the sample code this is done by reconstructing the values for the n and w inputs from the array indices. In a solution based on client indexation this may be accomplished by recovering a copy of the saved parameters.

Sometimes, optimized signaling schemes can be obtained by a careful analysis of the interplay between the methods executing `signal()` and the formulas associated to each condition queue. Figure 14 shows an optimized version of the monitor implementation for readers and writers seen before. The code has been obtained from the table in Figure 15. The cell at the intersection between row i and column j contains a (*minimal*) formula ϕ_{ij} such that $\text{POST}(op_i) \wedge \phi_{ij} \Rightarrow W_j$. For instance, the first row shows that the only possible signal after executing *beforeRead* is on the *readersCond* queue, unconditionally. The third row shows that there is no possibility of signaling after executing *beforeWrite*. The case for *afterWrite* is problematic (fourth row), as after execution the CPRES for both *beforeRead* and *beforeWrite* hold. In this case, we pass the baton to readers after checking that there is actually some thread blocked (line 67) and otherwise a signal on *writersCond* is issued. The most interesting case is the second row. The right column requires an extra condition for the CPRE of *beforeWrite* to hold. Apparently, a conflict similar to the previous case should arise if we look at the left column. However, it can be proved that if the threads invoke the methods respecting the protocol *readersCond.waiting() = 0* is an invariant at that point. Consequently, the only useful signal is on *writersCond* (line 35).

3.2.3 Client Indexation

When the set of equivalence classes is not so “small” or is impractical to be handled using one condition queue for each representative, a new approach should be followed.

```

1 public class ReadersWritersMonitorOpt {
2
3     protected int readers = 0, writers = 0;
4     protected Monitor mutex = new Monitor();
5     protected Cond writersCond = mutex.newCond();
6     protected Cond readersCond = mutex.newCond();
7
8     public void beforeRead() {
9         //@assume true;
10        mutex.enter();
11        if (writers > 0) readersCond.await();
12
13        /*@ assert true && \invariant_for(this)
14           @    && writers == 0;
15           @*/
16        readers++;
17        /*@ assert readers > 0 && \invariant_for(this);
18
19        readersCond.signal();
20        /*@ assert readers > 0 && \invariant_for(this);
21        mutex.leave();
22    }
23
24    public void afterRead() {
25        //@ assume readers > 0;
26        mutex.enter();
27        /*@ assert readers > 0 && \invariant_for(this)
28           @    && true;
29           @*/
30
31        --readers;
32        /*@ assert readers >= 0 && \invariant_for(this);
33
34        // checking for current state
35        if (readers == 0) writersCond.signal();
36        /*@ assert readers >= 0 && \invariant_for(this);
37        mutex.leave();
38    }
39
40    public void beforeWrite() {
41        //@ assume true;
42        mutex.enter();
43        if (!(readers == 0 && writers == 0))
44            writersCond.await();
45
46        /*@ assert true && \invariant_for(this)
47           @    && readers == 0 && writers == 0;
48           @*/
49        writers++;
50        /*@ assert writers==1 && \invariant_for(this);
51
52        mutex.leave();
53    }
54
55    public void afterWrite() {
56        //@ assume writers = 1 ;
57        mutex.enter();
58        if (!true) writersCond.await();
59
60        /*@ assert writers==1 && \invariant_for(this)
61           @    && true;
62           @*/
63        writers--;
64        /*@ assert writers==0 && \invariant_for(this);
65
66        int signaled = 0;
67        if (readersCond.waiting() > 0) {
68            readersCond.signal();
69            signaled++;
70        } else {
71            writersCond.signal();
72            signaled++;
73        }
74        /*@ assert signaled >= 0 && signaled <= 1;
75        /*@ assert writers==0 && \invariant_for(this);
76        mutex.leave();
77    }
78 }

```

Fig. 14 Readers & Writers implementation using priority monitors with optimized signaling code.

cond. queue signaling method	readersCond	writersCond
BR	true	false
AR	true*	readers = 0
BW	false	false
AW	true	true

Fig. 15 Signaling table for the readers & writers monitor.

```

1 private Monitor mutex;
2 private Map<Set, Monitor.Cond> conditionsOpi;
3
4 public TN opi(T1 arg1, ..., TM argM)
5     throws PreViolationResourceException{
6
7     mutex.enter();
8     if (!Pred2Java(PREopi(a1, ..., an))) {
9         mutex.leave();
10        throw new PreViolationResourceException();
11    }
12    //@ assume Pred2JML(PREopi(a1, ..., an));
13    TN result = null;
14    if (!Pred2Java(CPREopi(a1, ..., an))) {
15        Monitor.Cond c = mutex.newCond();
16        conditionsOpi.put(new Set(a1, ..., an), c);
17        c.await()
18    }
19    /*@ assert Pred2JML(PREopi(a1, ..., an))
20       @    && \invariant_for(this)
21       @    && Pred2JML(CPREopi(a1, ..., an));
22       @*/
23    Post2Java(POSTopi(a1, ..., an));
24    /*@ assert Pred2JML(POSTopi(a1, ..., an))
25       @    && \invariant_for(this);
26       @*/
27
28    // signaling code
29    int signaled = 0;
30    for (each condition conditionsOpi && signaled == 0) {
31        for (Map.Entry<Set, Monitor.Cond> entry:
32            conditionsOpi.entrySet()) {
33            if (!Pred2Java(CPREopi(entry.getKey()))) {
34                conditionsOpi.remove(entry.getKey());
35                entry.getValue().signal();
36                /*@ assert Pred2JML(CPREopi());
37                signaled++;
38                break;
39            }
40        }
41    }
42    //@ assert signaled >= 0 && signaled <= 1;
43    /*@ assert Pred2JML(POSTopi(a1, ..., an))
44       @    && \invariant_for(this);
45       @*/
46    return result;
47 }

```

Fig. 16 Generic template for implementing a shared resource using the client indexation technique. With a slight notational abuse, we have used the outcome of a Java method call (`entry.getKey()`) to instantiate the value of one CPRE with a set of actual parameters. A similar “trick” is used in Fig. 25.

The generic template for this technique is illustrated in Figure 16. In this case, when a thread makes an operation call but the CPRE does not hold, the code block that call on a brand new condition c for that call (line 15). Then, if the CPRE depends on some parameters, say x , the pair $\langle x, c \rangle$ is added to a map data structure (line 16) associated to the call operation.¹⁰

After the code of the method obtained from the POST, the unblocking code must signal a invalid blocked thread. For that, an iteration over all the data structures associated to any method is performed seeking for a “signable” thread (a blocked call which associated CPRE holds). Thus, for each pair presented in the map, the code check the corresponding method CPRE using the store parameters (line 33) holds. If it holds, the pair is removed from the map (line 34) and the thread blocked in the condition is signaled (line 35). Notice that the “0-1” schema must be accomplished in this template.

Shipping Warehouse Implementation (by Client Indexation)

Figure 17 outlines the implementation of the `exitWarehouse` operation using client indexation. In this case, we have inverted the map by using `Monitor.Cond` objects as key instead of the number of warehouse for practical reasons¹¹ (see line 10).

The rest of the implementation is quite straightforward. When the CPRE of a call does not hold, a brand new condition is created (line 17) and then stored in the map with the warehouse id (needed to check `exitWarehouse` CPRE).

The unblocking code differs slightly from the Figure 13. The iteration over the condition queues instantiated for the `enterWarehouse` remains untouched but the second loop changes a little bit. For each entry, the id of the warehouse is recovered and the CPRE is checked (lines 54-55). If it holds, there is a thread blocked in the condition that must be awakened (line 57). Notice that it is mandatory to remove the pair from the map and that there is no need to check the existence of the thread.

```

1 private Monitor mutex;
2
3 // Resource state
4 private int weight[];
5 private boolean occupied[];
6
7 // Handling concurrency
8 private Monitor mutex;
9 private Cond enteringWarehouse[];
10 private Map<Monitor.Cond, Integer>
11     conditionsExiting;
12
13 public void exitWarehouse(int n, int w) {
14     mutex.enter();
15
16     if (n != N_WAREHOUSES -1 && occupied[n]) {
17         Monitor.Cond cond = mutex.newCond();
18         conditionsExiting.put(cond, n);
19         cond.await();
20     }
21
22     /*@ assert warehouse == N_WAREHOUSES -1
23        @      || !corridor[warehouse];
24        @*/
25
26     weight[n] -= w;
27     if (n != N_WAREHOUSES -1) occupied[n] = true;
28
29     dummyUnblockingCode();
30
31     mutex.leave();
32 }
33
34 private void dummyUnblockingCode(){
35     bool signaled = false;
36
37     // trying to unblock a robot that wants to enter
38     ...same piece of code from Figure 13 (lines 51-65)
39
40     if(! signaled)
41         // trying to unblock a robot that wants to
42         // leave the warehouse...
43         // for each warehouse...
44         for(Entry<Monitor.Cond, Integer> entry:
45             conditionsExiting.entrySet()){
46             if ( entry.getValue() == N_WAREHOUSES -1 ||
47                 occupied[entry.getValue()] ) {
48                 conditionsExiting.remove(entry.getKey());
49                 entry.getKey().signal();
50                 /*@ assert entry.getValue() == N_WAREHOUSES -1
51                    @      || !corridor[entry.getValue()];
52                    @*/
53                 signaled = true;
54                 break;
55             }
56         }
57     }
58 }

```

Fig. 17 Warehouse implementation using client indexation technique.

3.3 Model-based Code Generation Using JCSP

The following paragraphs introduce two more templates for translating a shared resource specification into a Java class implementation using the JCSP library [29]. For each proposal, we follow the same approach as with the already presented monitor-based templates.

¹⁰ If the CPRE is independent, the code only stores the new condition queue in a method associated list.

¹¹ A map associates a specified value with a specified key in the structure. If the map previously contained a mapping for the key, the old value is replaced by the specified value.

This third-party library provides an implementation of CSP¹², the well-known theory for specifying and verifying complex patterns of behavior arising from interactions between concurrent objects. It encapsulates fundamental principles of communication by yielding sub-hundred-nano-second overheads for context switching, process startup/shutdown, synchronized channel communication and high-

¹² Another popular CSP implementation is *Communicating Threads in Java (CTJ)* [18]. There is also available a detailed comparison between JCSP and CTJ in terms of their philosophies, similarities and differences, performance, and usage scenarios in [28].

level shared-memory locks. The main components of CSP are messages and channels. While a message is just any object (even a channel), a channel is a model for synchronization. A message may be sent over a channel, and another process is able to read the messages sent over a channel it has a reference to. Those channels are inherently synchronous (rendezvous behavior), i.e. a thread waiting to receive a message from a channel will block until the message is sent.

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 *CSPProcess* 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.

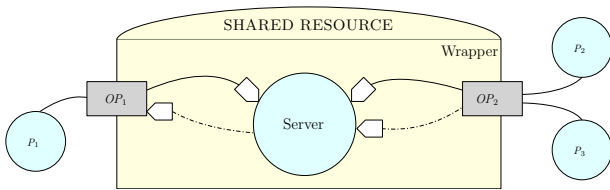


Fig. 18 Graphical representation of our message passing approach: the wavy lines indicate channels permanently present; the dotted lines indicate channels created on demand by an operation method, if needed, and only used during the operation.

This architecture is shown graphically in Figure 18. 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.

```

1 class ReadersWritersCSP implements CSPProcess,
2     ReadersWriters {
3
4     protected final Any2OneChannel chBeforeWrite;
5     protected final Any2OneChannel chBeforeRead;
6     protected final Any2OneChannel chAfterWrite;
7     protected final Any2OneChannel chAfterRead;
8
9     protected int readers = 0;
10    protected int writers = 0;
11
12    // interface ReadersWriters
13    public void beforeWrite() {
14        chBeforeWrite.out().write(null);
15    }
16    public void afterWrite() {
17        //@ assume writers == 1;
18        chAfterWrite.out().write(null);
19    }
20    public void beforeRead() {
21        chBeforeRead.out().write(null);
22    }
23    public void afterRead() {
24        //@ assume readers > 0;
25        chAfterRead.out().write(null);
26    }
27
28    // the server
29    public void run() {
30        // One entry for each method
31        Guard[] guards = {chBeforeWrite.in(),
32                        chBeforeRead.in(),
33                        chAfterWrite.in(),
34                        chAfterRead.in()};
35        final int BEFORE_WRITE = 0, BEFORE_READ = 1,
36              AFTER_WRITE = 2, AFTER_READ=3;
37
38        // conditional reception
39        boolean syncCond[] = new boolean[4];
40        Alternative services = new Alternative(guards);
41        while (true) {
42            syncCond[BEFORE_WRITE] =
43                (readers == 0 && writers == 0);
44            syncCond[BEFORE_READ] = writers == 0;
45            syncCond[AFTER_WRITE] = true;
46            syncCond[AFTER_READ] = true;
47
48            int service = services.fairSelect(syncCond);
49            switch(service){
50                case BEFORE_WRITE:
51                    chBeforeWrite.in().read();
52                    //@ assert writers == 0 && readers == 0;
53                    writers++;
54                    break;
55
56                case BEFORE_READ:
57                    chBeforeRead.in().read();
58                    //@ assert writers == 0;
59                    readers++;
60                    break;
61
62                case AFTER_WRITE:
63                    chAfterWrite.in().read();
64                    writers--;
65                    break;
66
67                case AFTER_READ:
68                    chAfterRead.in().read();
69                    readers--;
70                    break;
71            }
72        } // end while
73    } // end run
74 }

```

Fig. 19 Readers & Writers implemented using JCSP.

Applying these ideas to a simple resource like the Readers/Writers (Figure 3) 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 19.

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` could be slightly optimized to avoid reassigning positions with constant values (i.e. deleting lines 45 and 46). 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.

In our model-based approach this decision is, again, made after a logical analysis of the CPRES. From that, four cases can be distinguished:

I) The CPRE is `true`. Thus, a unique 'always-open' channel per operation is required and all messages are processed when the server receive them.

II) The CPRE is a formula independent from the operation's inputs. In this case, a single channel is also needed but its status depends on the inner state of the resource.

However, as explained for the monitor-based template, such a simple scheme is not always possible. A more sophisticated protocol is then needed for the server to decide when a request may be processed. That is why we present some variation of the previous two cases.

III) [CHANNEL REPLICATION] When CPRE formula depends on some input for the operation but the set of equivalence classes induced is finite (and relatively small), a channel for each equivalence class is created. If the CPRE is evaluated to `true` at call time, then the channel will be open for the server point of view.

IV) [DEFERRED REQUESTS] When CPRE depends on some input but the set of equivalence classes induced is either infinite or too big, a unique channel for the operation

is used. Every call is translated into a message including a fresh channel (allowing the communication between the server and the invoker) and the input data needed to check the associated CPRE.

The latter two styles are fully described in the following sections¹³ by showing how the CPRE can be systematically treated.

3.3.1 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, reconsider the *Shipping Warehouse* specification (Figure 6). As it is stated before, CPRES depend on values obtained from their parameters (the number of the warehouse and the robot's weight).

Generic Template 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* and the generic template for this approach is presented in Figure 20.

Considering the set E_i of equivalence classes induced by the parameter x of the operation op_i as is expressed in the Definition 1. If E_i is finite, then one channel can be defined for each equivalence class (as is represented by the conditions list in line 1). 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 both, as required above (line 8).

Due to the signature of `fairselect` CSP operation, these channels will have to be placed somewhere inside the overall array of channels of the server process as is shown in line 23. This leads to the following definitions:

Definition 2 Consider one operation op_i that takes arguments x and $\bar{y}$. so that its CPRE C_i depends on x . Let D_x, D_y be the domains of $x, \bar{y}$ and E_x the set of equivalence classes induced by the parameter x . As E_x is finite, there **must** be a partial injective function

$$pe_i : E_x \rightarrow \mathbb{N}$$

¹³ The resemblance with the monitor templates should be evident. We are using synchronous channels to emulate the functionality provided by condition queues.

```

1 private final List<Any2OneChannel> channels;
2
3 /* WRAPPER IMPLEMENTATION */
4 public TN opi(T1 arg1, ..., TM argM)
5     throws PreViolationResourceException{
6     //@ assert Pred2JML(PREopi(a1,...,an));
7     One2OneChannel chResp = Channel.one2one();
8     channels[pei(pxi(arg1,...,argr))] .out().write(
9         new Request(chResp.out(), arg1,...,argr));
10
11     //@ assert Pred2JML(CPREopi(a1,...,an));
12     TN res = (TN)chResp.in().read();
13     //@ assert Pred2JML(POSTopi(a1,...,an)) && \↦
14         invariant_for(this);
15     return res;
16 }
17
18 /* SERVER IMPLEMENTATION */
19 private static final int METHODiXl = fun(i,l);
20 ...
21 //@ Object ghost params;
22 public void run() {
23     //Union of all method's associated channels
24     Guard[] inputs={...methodiXlChannel.in(),...};
25
26     // Conditional reception for fairSelect().
27     //@ assert inputs.length == K+channels.size();
28     boolean syncCs[] = new boolean[K+channels.size()];
29     ...initialized syncCs array
30
31     final Alternative services =
32         new Alternative(inputs);
33     int service;
34
35     while (true) { // Server loop
36         // refreshing synchronization conditions
37         ...updating syncCs array
38         //@ assert (\forall int i;
39             @ i >= 0 i < syncCs.length;
40             @ syncCs[i] ==> C(i));
41         @*/
42
43         service = services.fairSelect(syncCs);
44         /* assume service < guards.length &&
45             @ service >= 0 && syncCs[service] &&
46             @ guards[service].pending() > 0;
47             @*/
48         switch(choice){
49             ...
50             // receiving and processing the request
51             case METHODiXl:
52                 //@ set params = pxi-1(pei-1(METHODiXl));
53                 /* assert Pred2JML(PREopi(params))
54                     @ && Pred2JML(CPREopi(params));
55                     @*/
56                 Request request = ((Request)
57                     ((AltingChannelInput) inputs[METHODiXl])
58                     .read());
59                 TN result;
60                 PostJava(POSTopi(params));
61                 /* assert Pred2JML(POSTopi(params)) &&
62                     @ \invariant_for(this);
63                     @*/
64                 request.getChannel().out().write(result);
65                 break;
66             }
67 } // end while
68 } // end run

```

Fig. 20 Generic template for implementing a shared resource using the channel replication technique. Channel definition based on the domain of the input parameters.

```

9 // Resource state
10 private int weight[]; // # = N_WAREHOUSES
11 private boolean occupied[]; // # = N_WAREHOUSES
12
13 /** WRAPPER IMPLEMENTATION */
14 // # = N_WAREHOUSES X MAX_WEIGHT
15 Any2OneChannel chEnteringWarehouse[] [];
16 // # = N_WAREHOUSES
17 Any2OneChannel chExitingWarehouse[];
18
19 public WarehouseAccessControlCSP(int n, int w) {
20     ... initialize state and create channel
21 }
22
23 public void enterWarehouse(int n, int w) {
24     //@ assume n >= 0 && n < N_WAREHOUSES && w > 0;
25     chEnteringWarehouse[n][w-1].out().write(
26         new WeightedRequest(n,w));
27 }
28
29 public void exitWarehouse(int n, int w) {
30     //@ assume n >= 0 && n < N_WAREHOUSES && w > 0;
31     chExitingWarehouse[n].out().write(w);
32 }

```

Fig. 21 Warehouse implementation using the channel replication technique. Channel definition based on the domain of the input parameters. Wrapper (client side) operations.

Therefore, by construction of E_x , there must also exist a (total) injection

$$px_i : D_x \rightarrow E_x$$

These functions are used in the following way: pe_i maps an integer (array position) to each element of the set equivalence class while px_i helps us in select which channel to send the message through. 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 (line 8). The server is responsible for enabling that channel based on the values of the inner resource. This is accomplished by the update made over the syncCs array, making this piece of code critical due to the fact that enabling an erroneous channel could lead to a system failure.

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.

Shipping Warehouse Implementation (by Replicated Channels) Figure 21 outlines an implementation of the shipping warehouse resource using the channel replication technique.

Following the same analysis of the warehouse operations CPRE presented in Section 3.2, enteringWarehouse oper-

```

48 public void run() {
49     /*
50     * inputs[0..N_WAREHOUSES] -> exitWarehouse
51     * inputs[N_WAREHOUSES..] -> enterWarehouse
52     */
53     Guard[] inputs = new Guard
54         [N_WAREHOUSES + (MAX_WEIGHT * N_WAREHOUSES)];
55     for (int n = 0; n < N_WAREHOUSES; n++) {
56         inputs[n] = chExitingWarehouse[n].in();
57         // flattening enteringWarehouse channels
58         for (int w = 0; w < MAX_WEIGHT; w++) {
59             inputs[N_WAREHOUSES + n*MAX_WEIGHT+w] =
60                 chEnteringWarehouse[n][w].in();
61         }
62     }
63
64     Alternative services = new Alternative(inputs);
65     ...
66
67     boolean[] syncCond = new boolean
68         [N_WAREHOUSES + (MAX_WEIGHT * N_WAREHOUSES)];
69     while (true) {
70         // updating syncCond for enterWarehouse(0,w)
71         for (int n = 0; n < N_WAREHOUSES; n++) {
72             System.out.println(n);
73             //exitWarehouse(n,w) CPRE
74             syncCond[n] = !
75                 (n != N_WAREHOUSES) || !occupied[n];
76             // enterWarehouse(n,w) CPRE
77             for (int w = 0; w < MAX_WEIGHT; w++) {
78                 syncCond[N_WAREHOUSES + n*MAX_WEIGHT+w] =
79                     weight[n] + (w+1) <= MAX_WEIGHT;
80             }
81         }
82         ...
83
84         int service = services.fairSelect(syncCond);
85         ...
86
87         if (service < N_WAREHOUSES) { // exit OP
88             //@ assert service != N_WAREHOUSES ==> !←
89                 occupied[service];
90             ... make POST for exitWarehouse hold
91         } else { //service >= N_WAREHOUSES ==> enter
92             //@ assert weight[service] + w <= MAX_WEIGHT;
93             ... make POST for enterWarehouse hold
94         }
95     } // end of server loop
96 } // end of run method

```

Fig. 22 Warehouse implementation using the channel replication technique. Server component code.

ation depends on both parameters: n (id of the warehouse which finite domain is D_n) and w (weight of the robot with finite domain is D_w). Therefore, `enteringWarehouse` can be dealt with by sending its message down an appropriately chosen element from the matrix of channels (`chEnteringWarehouse`, position (n, w)). Hence, requests that it cannot currently handle can be refused (i.e. blocked).

In the case of `exitWarehouse`, it only depends on the parameter n . Therefore, these operations can be handled similarly to the previous one, by sending a message down one of a different array of channels (`chExitingWarehouse`).

Figure 22 outlines an implementation of the process servicing all the requests. The channels are put into a Guard array (ls. 55-62) according to the pe_i functions:

$$\begin{aligned}
 pe_{ew}(n, w) &= n \\
 pe_{xw}(n, w) &= N_WAREHOUSES + n * MAX_WEIGHT + w
 \end{aligned}$$

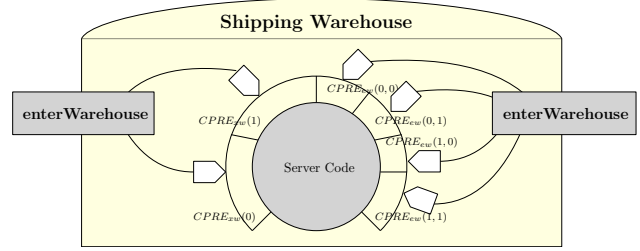


Fig. 23 Graphical representation of the syncCond array for the warehouse problem with two warehouses ($N_WAREHOUSES == 2$) and a maximum weight of two ($MAX_WEIGHT == 2$). Starting from the left corner, the first two positions are used for `enterWarehouse` CPRE whilst the other for `exitWarehouse` CPRE.

Notice that pe_{ew} is the flattening function that maps each position of the matrix of channels to the array passed as argument to `fairSelect`. Figure 23 illustrate pe and px graphically.

The JCSP Alternative is constructed with these guards (line 64), which will be `fairSelected` using the syncCond array of pre-conditions (line 67). Since these pre-conditions depend on the inner state of the shared resource (current weight in each warehouse and the robots present in the corridors), they are computed at the start of each service loop (lines 71-81).

The fact that the pe_i functions have disjoint ranges ensures the existence of an inverse mapping from $\mathbb{N}$ to $\{enterWarehouse, exitWarehouse\}$ that is used to translate the value returned by `fairSelect` into the original operation (line 88) and, then, to recover the original parameters by taking the inverses px_{ew}^{-1} and px_{xw}^{-1} (lines 93 and 99).

3.3.2 Deferred Requests

One restriction that is imposed by the previous template, is the necessity of having an “small” finite set of equivalence classes of all calls to some resource operation, modulo logical equivalence of their CPRES. Even when those set are relatively small in size, the definition of the mappings ($pe_i, px_j \dots$) involved could be error-prone. As an alternative to the previously presented schema, we introduce a new template called *deferred requests* that does not suffer from these problems. As we will see, the only requirement is that the number of clients is finite.

Generic Template In Figures 24 and 25 the generic template of the Wrapper and Server Components, respectively, for this approach are presented. The core idea is to have a unique channel per operation as can be seen in lines 1-3. As explained before, this implies that if an operation CPRE depends on a parameter, the condition cannot be encoded in the corresponding reception list. Hence, the method channel is

```

1 private Any2OneChannel chOperation1;
2 ...
3 private Any2OneChannel chOperationM;
4
5 /*
6  * single send: PRE depends on the actual
7  * parameters and on the resource state
8  */
9 public TN opi(T1 arg1, ..., TM argM)
10     throws PreViolationResourceException{
11     /* if PREopt(a1,...,an) depends on inner state of the
12      * resource, server component has to check it
13      * before deferring the call
14      */
15     One2OneChannel chResp = Channel.one2one();
16     OpIPetition pet =
17         new OpIPetition(arg1,...,argN-1,chResp.out());
18
19     // sending petition to the server
20     chOperationi.out().write(pet);
21
22     // waiting for notification from the server
23     Object obj = chResp.in().read();
24
25     if(obj instanceof PreViolationResourceException){
26         throw (PreViolationResourceException) obj;
27     }
28
29     /*@ assert Pred2JML(POSTopt(a1,...,an))
30      @      && \invariant_for(this);
31      @*/
32     return (TN) obj;
33 }
34
35 /*
36  * double send: when the footprint does not
37  * contain all parameter information.
38  */
39 public TN opi(T1 arg1, ..., TM argM)
40     throws PreViolationResourceException{
41     One2OneChannel chResp = Channel.one2one();
42     OpIPetition pet =
43         new OpIPetition(argk,...,argN,chResp);
44
45     // sending footprint to the server
46     chOperationi.out().write(pet);
47
48     /*@ assert Pred2JML(PREopt(a1,...,an))
49      @      && Pred2JML(CPREopt(a1,...,an))
50      @      && \invariant_for(this);
51      @*/
52     // sending the rest of the parameters
53     chOperationi.out().write({arg1,...,argk-1});
54
55     // waiting for notification from the server
56     Object obj = chResp.in().read();
57
58     /*@ assert Pred2JML(POSTopt(a1,...,an))
59      @      && \invariant_for(this);
60      @*/
61     return (TN) obj;
62 }
63 }

```

Fig. 24 Generic Wrapper Component of the deferred requests template.

always open but the wrapper send a *request* instead of a simple message (lines 15-20, 43-55). It must contain the *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.

```

68 /* SERVER IMPLEMENTATION */
69 /** for each method, a list for deferring
70  * the requests is created */
71 final Queue<OperationIPetition> operationIRequests;
72 .
73 .
74 final Queue<OperationNPetition> operationNRequests;
75 .
76 /** Constants representing API method's */
77 private static final int OPERATION_1 = 0;
78 .
79 private static final int OPERATION_M = M-1;
80
81 public void run() {
82     /* One entry for each method. */
83     Guard[] inputs = { chOperation1, ...,
84                       chOperationM};
85     Alternative services = new Alternative(inputs);
86
87     while (true) {
88         int service = services.fairSelect();
89
90         // receiving a petition from the wrapper
91         switch (service) {
92             ...
93             case OPERATION_1:
94                 OperationIPetition request =
95                     (OperationIPetition)
96                     chOperation1.in().read();
97                 operationIRequests.offer(request);
98                 break;
99             ...
100         }
101
102         // processing all possible requests
103         boolean anyResumed;
104         do {
105             anyResumed = false;
106             for (each list operationIRequests){
107
108                 int lastItem = exitRequests.size();
109                 for (int i = 0; i < lastItem; i++) {
110
111                     OperationIPetition curr =
112                         operationIRequests.pull();
113                     if (Pred2Java(CPREopt(curr.a1,...,curr.an))){
114
115                         Post2Java(POSTopt(curr.a1,...,curr.an));
116                         /*@ assert Pred2JML(POSTopt(curr.a1,...,curr.an))
117                          @      && \invariant_for(this);
118                          @*/
119
120                         // send a result back down the channel
121                         curr.getChannel().write(result);
122                         anyResumed = true;
123                     } else {
124                         operationIRequests.offer(curr);
125                     }
126                 }
127             }
128             // end for each list of requests
129             ... JML asserts that the CPRES for all the items left in
130             ... all deferred requests lists are false
131         } while (anyResumed);
132     } // end server loop
133 } // end run

```

Fig. 25 Generic Server Component of the deferred requests template.

Shipping Warehouse Implementation (by Deferred Requests)
Figure 26 outlines the implementation of the *wrapper* component operations for the warehouse example using the aforementioned technique. The message sent consists of a fresh channel (lines 21 and 31) bundled together with a *footprint* of the method parameters that contains sufficient informa-

```

18 public static final int N_WAREHOUSES;
19 public static final int MAX_WEIGHT;
20 private int weight[];
21 private boolean occupied[];
22 ...
18 public void enterWarehouse(int n, int w) {
19     EnterWarehousePetition pet =
20         new EnterWarehousePetition(n, w,
21             Channel.one2one());
22     // sending petition to the server
23     chEnterWarehouse.out().write(pet);
24     // waiting for notification
25     pet.getChannel().in().read();
26 }
27
28 public void exitWarehouse(int n, int w) {
29     ExitWarehousePetition pet =
30         new ExitWarehousePetition(n,
31             Channel.one2one());
32     // sending petition to the server
33     chExitWarehouse.out().write(pet);
34     // sending the weight
35     pet.getChannel().out().write(w);
36     // waiting for notification
37     pet.getChannel().in().read();
38 }
...

```

Fig. 26 Warehouse implementation using the deferred requests technique using only two channels. Wrapper (client side) operations.

tion for the server to compute the relevant CPRE. Depending on the content of the footprint sent, two cases can be distinguished:

- *single send*: when it contains all the actual parameter (e.g. the `enterWarehouse` operation, line 19), 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 `exitWarehouse` operation, line 33), having to send the rest of the formal parameter through the already created channel (line 35).

Notice that in both cases, the client will block its execution until a message (the return value or a meaningless token) is written into the invocation associated channel notifying that the request was processed successfully (lines 25 and 37 of Figure 26).

Nevertheless, the tricky part comes when designing the server code (Figure 27). This component is a sequential process that defers all requests into some auxiliary data structure, to be processed later. In order to avoid errors, 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.

Considering the first point, requests are not served in the same order as they are processed by the server, so the implementation of component has the responsibility for ensuring fairness. 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).

Getting back to the warehouse example, the reception of messages in the server code is straightforward. In the implementation, the requests are simply put in a list – one for each operation. Figure 27 shows part of the server code component for treating the requests.

The basic structure of the code is a set of repetitive sentences. The first loop (line 102) unblocks `exitWarehouse` requests whilst the second (line 134) processes all `enterWarehouse` petitions. Usually, looping over the two lists of deferred requests will be enough to guarantee that all reminder requests cannot proceed. However, in some cases, there can be a ‘ping pong’ effect between operations, i.e. serving a request from one operation can enable a petition that was already checked (its CPRE did not hold with the state at that time).

A possible scenario for a ‘ping-pong’ effect is when a robot *Y* is waiting for leaving warehouse 0 (`exitWarehouse(0, wy)`) and another robot *X* is waiting to enter to the warehouse 1 (`petition enterWarehouse(1, wx)`) that makes that corridor 0 is occupied, `corridor[0]` is true. When the server has already processed all `exitWarehouse` petitions (first loop), the robot *Y* is still blocked due to the fact that the next corridor is occupied by the robot *X* (`corridor[0]` holds). Then, the server continues its execution by processing all `enterWarehouse` invocations and, in particular, it processes the petition `enterWarehouse(1, wx)` of the robot *X*, setting `corridor[0]` to *false*. After that second loop, the state of the server does not meet the second requirement of the three presented above for a correct implementation – because the CPRE of the petition of the robot *Y* holds and has not been processed. That is why, we have to enclose the two request processing loops in another loop (lines 99-174) that exits only when none of their CPRES are true (handled through the `anyResumed` flag). For verification (Section 3.2.1), asserts are added to that effect are made following that enclosing loop.

```

30 private final List<EnterWarehousePetition>
31     enterRequests = new ArrayList<>();
32 private final List<ExitWarehousePetition>
33     exitRequests = new ArrayList<>();
34 ...
68 private static final int ENTER = 0, EXIT = 1;
69
70 public void run() {
71     /* One entry for each method. */
72     Guard[] inputs = { chEnterWarehouse.in(),
73                       chExitWarehouse.in() };
74     Alternative services = new Alternative(inputs);
75
76     while (true) {
77         int service = services.fairSelect();
78         // synchronization not checked here!
79         switch (service) {
80             case ENTER:
81             EnterWarehousePetition enterRequest =
82                 (EnterWarehousePetition)
83                 chEnterWarehouse.in().read();
84             enterRequests.add(enterRequest);
85             break;
86             case EXIT:
87             ExitWarehousePetition exitRequest =
88                 (ExitWarehousePetition)
89                 chExitWarehouse.in().read();
90             exitRequests.add(exitRequest);
91             break;
92         }
93         ...
94
95         boolean anyResumed;
96         do {
97             anyResumed = false;
98             int lastItem = exitRequests.size();
99             for (int i = 0; i < lastItem; i++) {
100                 ...dequeue an ExitWarehousePetition from exitRequests
101                 if (curr.getWarehouse() == N_WAREHOUSES ||
102                     !occupied[curr.getWarehouse()+1]) {
103                     /*@ assert
104                      @ curr.getWarehouse() == N_WAREHOUSES ←
105                      ||
106                      @ !occupied[curr.getWarehouse()+1];
107                      @*/
108                     ...make POST of exitWarehouse hold
109                     // send a null back down the channel
110                     curr.getChannel().out().write(null);
111                     anyResumed = true;
112                 } else {
113                     ...enqueue curr back in exitRequests
114                 }
115             }
116
117             lastItem = enterRequests.size();
118             for (int i = 0; i < lastItem; i++) {
119                 ...dequeue an EnterWarehousePetition from enterRequests
120                 if (curr.getWeight() +
121                     weight[curr.getWarehouse()] <= MAX_WEIGHT
122                 ){
123                     /*@ assert curr.getWeight() +
124                      @ weight[curr.getWarehouse()]
125                      @ <= MAX_WEIGHT;
126                      @*/
127                     ...make POST of enterWarehouse hold
128                 } else {
129                     ...enqueue curr back in enterRequests
130                 }
131             }
132
133             ... JML asserts that the CPRES for all the items left in
134             ... enterRequests and exitRequests are false
135         } while (anyResumed);
136     } // end server loop
137 } // end server

```

Fig. 27 Shipping Warehouse implementation using deferred requests. Server side message processing.

4 Verification

Applying the templates presented in the previous section is not a fully automatic process. While the templates provide a syntactic structure, ensuring that the actual code fragments that “fill in” the holes in the template comply with the corresponding proof obligations is still the programmer’s responsibility. The goal of the verification process we are presenting in this section is to prove (mechanically) that the code obtained by application of some template – which we assume to adhere to its structure, syntactically – also complies with the proof obligations required. As we will show in the following paragraphs, doing this, even with the help of a mature verification environment for Java is a challenging task. However, with this “template-driven” approach to verification, we manage to replace a much harder verification problem – i.e. to fully verify the behaviour of the Java class against a transition semantics of the shared resource – by a set of smaller verification challenges, once per proof obligation.

We have chosen KeY [6] because it supports Java code with JML annotations and is continuously maintained offering a solid verification framework that suites perfectly the purposes of this paper¹⁴. 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 *PRE* (*requires* JML clause), postcondition *POST* (*ensures* clause) and method modifiers (*assignable* clause). KeY tries to prove that the execution of an operation, in a state in which class invariant *I* and the precondition *PRE* holds, will terminate normally and in a state in which the postcondition holds.

To accomplish the verification, we assume that the code under analysis adheres, at least, syntactically to the structure described in Section 3. This code is expected to contain logical (JML) annotations at certain points determined by the template logic (for instance, ln. 37 and 43 of Figure 21). Then, the code (Java and JML) needs to be instrumented for KeY to be able to reason about the concurrent interplay between the processes and the shared resource. This instrumentation helps KeY figure out what happens behind the scenes with the specific concurrent mechanisms used in the implementations (such as channels, priority monitors, etc.).

The following paragraphs present the verification process using a common structure for all the presented templates. First, we enumerate the proof obligations that the templates impose at relevant points of the program for both monitor (Section 3.2) and CSP-based implementations (Section 3.3). Then, we present how to instrument the generated

¹⁴ Other systems examined – and discarded for various reasons – were Slam [5], Bogor [26] and TACO [14].

code in order to make it verifiable by KeY. Finally, we give details on actually proving the properties in KeY.

4.1 Verification of Monitor-Based Implementations

The starting point of our framework is a set of proof obligations that are imposed by the resource interface (the specification) and the code template (the implementation scheme) at certain places. These are:

1. The *preconditions* (PRE) are often independent for state changes in the resource. In such cases, they should be checked at call time and throw the corresponding exception although for brevity, we have not included that in the example. However, they must be assumed before the relevant operations.
2. The *invariant* (INVARIANT) must have been established at object initialization time, must hold right before entering the core block (S), right before finishing method body execution, but also right before invoking `wait()`.
3. The *concurrent* or synchronization pre-condition (CPRE) must hold right before entering the core for each operation.
4. The *postcondition* (POST) must hold on return of each operation.
5. Whenever a `signal()` is invoked on a condition q , the safety assertion for that condition W_q can be established (*soundness of signals*).
6. If some thread has performed its operation and there is one (or more) thread blocked on some condition q such that its CPRE holds, then one of these threads must be signaled (*completeness of signals*).
7. At most one `signal()` can be issued during the execution of a monitor operation.

Properties 1–4 have to do with the observable behaviour of the shared resource, i.e. the service offered to the client threads and are, then, independent of the implementation method chosen. Properties 5–7 in this list, however, are specific to the implementation method (priority monitors in this case) and will be the ones we will pay more attention in the following paragraphs.

4.1.1 Instrumentation

As KeY does not offer support for concurrency or monitor primitives, we had to *instrument* our monitor library so as to provide KeY with a basic understanding of what is happening behind the scenes. For example, each condition is substituted by an `int` variable which will count the number of waiting threads on it, and therefore the signal operation just decreases that variable, etc. The codification of each property described below can be easily expressed as *ensure* clauses.

Soundness (`prop_cs_preservation`) Conditional synchronization (CPRE formula) of a method holds immediately before executing the method code. To complete the verification, we need to prove that when a method is blocked, it will be enqueue in the corresponding condition queue based on the equivalence class which belongs to as it was shown in Section 3.2. . The main difficulty in verifying this property comes from the issue that KeY cannot find any relation between the assertion of line 20 (Figure 13) and the program point immediately after the thread is unblocked(line 52).

Due to that, we rely on the following facts: (i) all signals are safe , (ii) only one `signal` is executed and (iii) the semantic of exclusion of priority monitors, i.e. there is no possibility of a spare thread reaching the monitor between executions. Hence, the proof is trivial without needing to perform any extra instrumentation.

Post-condition (`prop_signal_and_exit`) Even though a signal is executed or not, all variable representing the inner state of the shared resources must not change and preserves the Post-condition of the operation. That can be easily described by adding an assignable clause (ls.56-57) containing only the instrumentation variables.

The goal of this property is to avoid side effects on the unblocking code. Another approach to ensure this property would be to directly implement a *signal and return* [20] policy. Such policy would suit perfectly since it also helps in simplifying the verification of *prop_0_1_signal*.

Soundness of Signals (`prop_safe_signal`) The unblocking code must guarantee that when a `signal` is executed, the conditional synchronization of the awakened thread holds. This property ensures *safety* of the unblocking code present in each method. As can be seen in the unblocking code, we define an assert for each operation's CPRE as is shown in Figure 13.

Since KeY cannot easily interpret JML assertions, we define a *JML ghost field*¹⁵ called `awakenThread` that will contain the position of the array of the signaled condition queue. Initially, it must be `-1` and then updated if a signal was performed. Besides, we add an *ensure* clause depicting that if a signal is executed, the synchronization condition of the thread must hold as is shown in lines 55-68 of the appendix B.

This clause expresses that if a condition queue decreases its size, i.e a `signal()` is executed, then the associated CPRE of the method of the signaled thread must hold.

The aim of this property is to ensure that every time a thread is awakened, it is able to proceed with the execution,

¹⁵ A ghost field is similar to a JML model field, in that it is also only present for purposes of specification and thus cannot be used outside of JML annotations. .

i.e. its associated synchronization condition holds. The errors detected by this property are associated with a bad definition of the conditions array or an improper unblocking schema.

Completeness of Signals This is implemented using two properties in KeY:

(prop_signal_effective) The executed `signal` (if any) in the unblocking code is effective, i.e. it actually wakes a blocked thread up. This is visible by a decrement in size of the condition queue. It is expressed as the *ensure* clause in lines 94-106 of the appendix B.

This property partially verifies the correct behavior of the monitor implementation by checking that every time a thread is awakened from a condition queue, its size decreases in 1.

(prop_liveness) The current thread must not leave the monitor without signaling one *improperly blocked* thread, if it exists. Improperly blocked threads are those blocked in a condition queue when its CPRE already holds. The resulting *ensure* clause is shown in lines 108-126 of the appendix B.

The aim of the property is to avoid having improperly blocked threads without being signaled after the execution of the unblocking code.

At most 1 signal (prop_0_1_signal) Only one thread can be awoken in each execution. The unblocking code must guarantee this in order to forbid that several threads access to the critical section of the given shared resource at the same time.

Since we have defined `signaled` variable and we must accomplish “0-1 coding scheme”, the property

```
signaled >= 0 && signaled <= 1
```

is added as an instance invariant. Besides, it is mandatory to ensure that one and only one condition queue is decremented, unless any thread was awakened. In that case, we must verify that all condition queues are keeping their size by adding a JML *ensure* clause depicted in lines 70-92 expressing that only one condition queue decreases in size.

The goal of having this property is the detection of badly nested conditional or iterative sentences, allowing several executions of `signal` operation. A relevant piece of instrumented code of `WarehouseAccessControlMonitorNaive` class is shown in appendix B.

4.2 Verification of CSP-Based Implementations

4.2.1 Channel Replication

The implementation of the Warehouse resource enable the server to determine whether the relevant CPREs hold for all the operation request channels being serviced, 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 technique and verification of such implementations is presented in Section 4.2.2.

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 example), rather than `if/else` (used by the Warehouse server via channel replication).¹⁶

Instrumentation As we did for monitors, we proceed to instrument the behaviour of the concurrent mechanism, such as the channels. 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 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 as is shown in Appendix A. 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 correct *Method treatment* option) that make simpler to simulate JCSP method `fairSelect`. 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 Appendix A.

For the purposes of KeY verification, it does not matter that this adaptation of the server shuts down (unlike the actual servers in Figures 21 and 22).

As we did for the monitor-based implementations, we start by proposing a set of proof obligations for the server component of our code generation template using channel replication technique. Then, we show how to instrument them in order to be verifiable in KeY:

- 3_a. the CPRE must hold right before reading the request from the operation associated channel (*soundness of the server*);
- 5_a. the selection of the channel to read from must be open (its associated `syncCond` position must hold) and it must have a message waiting to be read (*correctness of the server*);
- 6_a. only one request can be processed in each server iteration (*completeness of the server*).

¹⁶ Modifying this generic template for the `if/else` structure is straightforward.

Soundness of Server Component (`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 22, it is asserted that CPRE holds just before updating the shared resource (line 93 for `enterWarehouse` operation and line 99 for `exitWarehouse`).

As was done before, a new class instance attribute, called `cprePreservation`, is created and initialized to `true` that will express this property. Every time a service is chosen, this accumulator variable is updated (conjunctively) with the associated CPRE. To be verifiable in KeY we added as a invariant of the server loop as well as an *ensure* clauses.

This expresses that when a method invocation is selected, processing of the request will not violate the CPRE of the method, i.e. no request is served in a state that does not satisfy its synchronization condition. Once the server knows which branch of the conditional sentence will be executed, `cprePreservation` is updated before processing the request as it is said before.

Correctness of Server (`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.

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 `syncCond` and (iii) their length must be the equal.

To make the assertions generated by the template (after updating `syncCond` array) verifiable in KeY, a new class instance boolean attribute `wellFormedSyncCond` is defined in order to check that each position of the array is consistent.

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.

Completeness of the Server (`prop_only_one_request`)

This property represents that exactly one request is processed per server iteration. This is already guaranteed if using nested `if` statements, but when using a `switch`, the execution of more than one branch is possible. The instrumentation for this property is rather simple: a counter variable `processedMessages` and a boolean variable `oneMessageProcessed` are created. 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.

4.2.2 Deferred Requests

Instrumentation In this case, 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 mechanism that impacts any variable 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.

The set of proof obligations for verifying the generated code based on deferred requests are:

- 3_b. CPRE must hold immediately after the server code retrieves a requests to be processed (*correctness of the server*)
- 6_b. 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 (*completeness*).

Correctness of the server (`prop_cs_preservation`) Immediately after the server code retrieves a request to be processed, the CPRE of the method associated with the request must hold. This is established with a JML *assert* before updating or accessing the resource – see Figure 27 (line 142 for the `enterWarehouse` operation and 109 for `exitWarehouse`). This restriction ensures *safety* of the processing code because changes to the state of the resources are performed only for those requests that represent valid invocations.

The verification of this property is extremely related with the homonym one from the previous template. To verify that this is always honored, a new class attribute is created and initialized to `true` and *anded* 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. The server then loops back to await further operation requests (as in Figure 27).

Completeness of the Server Component (`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 27 (immediately after line 174), we assert that for every request in the data structure (i.e. pending), their associated CPRE does not hold.

A request is either processed (if its CPRE holds) or enqueued again. If the former is true, the correctness of the server 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:

```
//@ ensures  $\bigwedge_{i=1}^n (\text{method}_i.\text{Requests} > 0 \implies \neg \text{CPRE}_i);$ 
```

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`:

```
/*@ ensures  $\sum_{i=1}^n \text{method}_i.\text{Request.size}() > 0 \implies \text{completeness};$ 
   @
   */
```

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.

5 Preliminary Results

In order to show that errors produced by an improper application of the schemata presented in the previous section can be detected by existing program verification tools, we have tested our approach on a number of examples.

A small set of problems have been considered. These include the readers and writers that we have used as running example as well as the warehouse 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. The growing set of examples, including specifications, documentation and (verified) code is accessible at [2]. Configuration files for reproducing the examples in KeY are included.

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

- Correct implementations of all 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 monitors, a better usage of conditions queues or an intelligent unblocking code. Considering 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
 - erroneous optimizations,
 - violations of *protocol* definitions.
 - erroneous or incomplete update of the `syncCond` array for those implementations based on channel replication.
 - missing `break` statements in `switch` statement for JCSP-based code;
 - incorrect optimizations on the code processing the pending requests, e.g. leaving requests unattended, or too strong conditions on these requests when using JCSP deferred requests technique;

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, it can try to figure out whether the formula presented is valid or not, or some rule can be applied so that the proof continues.

5.1 Monitor-based implementations

We have compared three implementations of the warehouse problem:

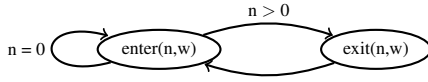


Fig. 28 FSM representing the behavior of the unblocking code of *Shipping Warehouse* example. An `enterWarehouse(n,w)` invocation can only wake up `enterWarehouse(0,w)` invocations (when $w=0$) or `exitWarehouse(n-1,w)` (when $w>0$). On the other hand, an `exitWarehouse(n,w)` can wake up `enterWarehouse(0,w)` invocations at any time and `enterWarehouse(n-1,w)` only when $n>0$.

- `WarehouseAccessControlMonitorNaive` class is a naive application of the template, thus, no optimization is made in the code. In this implementation, we applied the *parameter indexation* technique for both operations. For `enterWarehouse` operation, a $N_WAREHOUSES \times MAX_WEIGHT + 1$ matrix of condition queues is created and for `exitWarehouse` suffices with an array of $N_WAREHOUSES$ condition queues. The unblocking code simply loops over the whole matrix and array trying to find an illegally blocked thread, i.e a blocked thread which CPRE does hold.
- `WarehouseAccessControlMonitorOpt` class keeps using parameter indexation for the condition queues but has an intelligent unblocking code. Every time a thread finishes executing `enterWarehouse(n,w)`, it will try to wake up either an `enterWarehouse(0,w)` blocked invocation or an `exitWarehouse(n-1,w)` call (when $n > 0$). Considering `exitWarehouse`, the code will wake up a thread that is either waiting to enter the given warehouse or those that are waiting to enter the first warehouse. Figure 28 shows an automaton expressing the heuristic of the unblocking code.
- `WarehouseAccessControlMonitorBestOpt` class is an improvement of the previous one. For `exitWarehouse` operation, a simple array is still used. But, instead of using a matrix of condition queues for `enterWarehouse` invocations, we define simpler structures: (i) when $n > 0$, it suffices to declare an array of $N_WAREHOUSES$ of `WeightedConditions`, i.e condition queues that are associated with a determined weight; (ii) when $n = 0$, *thread indexation* technique is applied by using a `PriorityQueue` of `WeightedCondition` to gathers all blocked threads. Notice that in both cases, we need to store the weight of the call in order to check the CPRE before signaling the thread. The unblocking code differs slightly from the second implementation but follows the same principles expressed in Figure 28.

After following the proof obligations and the rules for instrumenting the monitor library stated in Section 4, we obtained the instrumented code for the naive implementation as shown in Appendix B.

As it can be seen in the first implementation, the unblocking code iterates over all the defined condition queues.

In each iteration, every blocked thread is checked against its associated synchronization condition even if it is already known that the associated condition does not hold. Then and only then, the code signals the correct condition queue. Having codified those clauses, the verification is possible in KeY and ends successfully, meaning that our naive unblocking code is correct.

Even though the solution is correct and satisfies the specification of the problem, this piece of code can be easily optimized. An invocation of `enterWarehouse(n,w)` can signal an `enterWarehouse(0,w)` invocation (when $w=0$) or `exitWarehouse(n-1,w)` (when $w > 0$). However, an `exitWarehouse(n,w)` can wake up either an `enterWarehouse(0,w)` call or `enterWarehouse(n-1,w)` when $n > 0$. Considering the previous description, we obtained the instrumented unblocking code for `enterWarehouse` operation shown in Appendix C.2¹⁷. Notice that the optimized code only iterates over those thread in which the associated weight will not exceed the constraint of the warehouse without checking directly its associated predicate (it is implicit by the starting point and invariant of the loop).

Both the elimination of one of the loops and the improvement of the other are nontrivial modifications and require the analysis of all possible scenarios. The KeY verification can detect errors easily due to the fact that we know beforehand that all *ensures* clauses of the previous implementation are correct and remain unmodified. If the prover stops, there must be some error in the optimization and the code needs to be revised. Otherwise, we are able to set the correctness of the performed optimization.

Once the second implementation is verified correctly and no errors were found, an optimization of the condition queues is made as described before. As a consequence of the redefinition of them, it is mandatory to rewrite all the *ensures* clauses. Once we have done all the modifications, we are able to verify the unblocking code in KeY by checking if the new structures satisfy the proof obligations depicted in Section 4. Finally, after carrying out all the verification process, we can conclude that the fully optimized version is correct.

5.2 JCSP-based implementations

Regarding the templates using JCSP, we only analyzed the code generated using the replication channel technique¹⁸. For that, we have conducted the same experiment over the

¹⁷ Notice that the unblocking code is now separate in two different pieces, one per each iteration. The code for `exitWarehouse` operation is analogous and can be seen in [2].

¹⁸ The experimentation over those implementations that used ‘deferred request’ techniques are not shown in this paper but downloadable from [2].

following implementations using JCSP of the warehouse problem:

- `WarehouseAccessControlMonitorCSP` class is a naive application of the template without performing any optimization. For `enterWarehouse` operation, a $N_WAREHOUSES \times MAX_WEIGHT + 1$ matrix of channels is created and for `exitWarehouse` suffices with an array of $N_WAREHOUSES$ channels. `syncCond` array is fully updated in each server iteration.
- `WarehouseAccessControlMonitorCSPOpt` class keeps the same principles as the former but the `syncCond` array is updated on those positions regarding the warehouse involved in the previous iteration. For instance, only the positions in correspondence with row n of `chEnterWarehouse` and `chExitWarehouse[n]` are updated if the last processed operation was `enterWarehouse(n, w)`.

As it can be seen at the beginning of the server loop in the first implementation, `syncCond` is fully updated, i.e. each position of the array is recalculated even if it is already known that a specific position does not change. Then, having codified the clauses explained in Section 4.2.2, the verification is possible in KeY and ends successfully, meaning that our simplest code is correct.

Even though the solution is correct and satisfies the specification of the problem, the server component can be easily optimized to gain efficiency (e.g., the portion of code in charge of updating `syncCond` array).

After an invocation of `enterWarehouse(n, w)`, the channels associated with any other warehouse but n remains in the same state (open or close) than before. That is, either the weight of the warehouse n has changed or the corridor $n - 1$ (if $n > 0$) is set free. Therefore, only those positions associated with the given warehouse (of both operations) must be updated¹⁹. Notice that all instrumentation regarding the proof obligations described in Section 4.2.1 remains unmodified, i.e. the variable `wellFormedSyncCond` should check that all positions of the array are consistent.

In some cases, improving the update code for `syncCond` is not trivial and require the analysis of all possible scenarios based on the previous processed request. The KeY verification can detect errors easily due to the fact that we know beforehand that all clauses of the previous implementation are correct and remain unmodified, as we did for the monitor-based template. If the prover stops, there must be some error in the optimization and the code needs to be revised. Otherwise, we are able to set the correctness of the performed optimization.

Table 1 summarizes the results showing whether the verification was successful or not, which template was applied

¹⁹ The analysis of the other scenario, when a `exitWarehouse` request is processed, is analogous to the presented one.

and how detailed is the output of KeY. In addition to the readers & writers and shipping warehouse running examples, the table includes a small set of extra examples.

The legend in the second column indicates which template was applied to the implementation. We add *MON* for those using priority monitors and *DF* and *CR* for the implementations based on JCSP library following ‘deferred requests’ or channel ‘replication technique’, respectively.

6 Related Work

Model-based methodologies are becoming increasingly popular in software development, specially in areas such as critical software, embedded systems, etc. The approach presented in this paper is part of a larger attempt to build a set of model-based techniques for producing correct concurrent software. See the works presented in [17], for model verification, and [13], for model-based testing, to name a few.

Framework similar in spirit exist also in the commercial domain. One example is Dezyne [9], developed by the embedded software company Verum (www.verum.com). It comprises a set of model-driven software engineering tools that helps developers formalize, create and verify pieces of software (sequential or concurrent). They present an overall approach starting from a “mental model” (as they called it) over which the general properties can be verified, such as a model being deadlock free, livelock free, and complying to its interface model, towards several techniques for automatic code generation. Our approach involves the translation of the Java annotated interface to TLA in order to check the aforementioned properties. Then, the methodology focuses on the design of templates to automatically generate code but at this moment, our templates cover just Java, Ada and Erlang languages as output. However, both approaches (ours and Verum’s) offer a strong traceability from models to code.

Also, Uppaal [22] offers a tool for modeling and verifying real-time systems using timed automata extended with integer variables, structured data types, user defined functions, and channel synchronization. It also offers a way of synthesizing the model of the controller component of code. When the model is verified, the user needs to resort to some third party tool for code generation like the one presented in [25]. Our approach presents support during the whole system development life cycle having different techniques for modeling, code generation and testing.

There is an increasing consideration of the usage of model-driven engineering approach for automatic code generation as can be seen in [21]. As we explained in the Introduction, there are practical (e.g. efficiency) reasons for letting the programmer retain the power to modify the code. However, our approach constrains the programmer’s freedom by

Example		Short description of the implementation	KeY Result
Readers & Writers	MON	correct implementation with readers priority	✓ verified
Readers & Writers	MON	WRONG implementation: a reader and a writer could proceed at the same time	✗ error detected: (readers == 1 & writers == 1)
Readers & Writers	MON	WRONG implementation: two readers can be awakened	✗ error detected: (signaled == 2)
Multibuffer	MON	correct implementation	✓ verified
Multibuffer	MON	WRONG implementation: allowing writing beyond the capacity of the buffer (bad CPRE codification)	✗ error detected: incomprehensible goal
Shipping Warehouse	MON	correct implementation	✓ verified
Shipping Warehouse	MON	WRONG implementation: unblocking code do not wake up robot with maximum weight	✗ error detected
Readers & Writers	DF	correct implementation with readers priority	✓ verified
Readers & Writers	CR	correct implementation with writers priority	✓ verified
Readers & Writers	DF	WRONG implementation: formula for proceeding <code>afterWrite</code> and <code>afterWrite</code> is weaker than expected allowing to have a false positive	✗ error detected: (readers == -1)
Readers & Writers	CR	WRONG implementation: missing <code>break</code> statement	✗ error detected: (oneMessageProcessed == false)
Shipping Warehouse	CR	correct implementation with readers priority	✓ verified
Shipping Warehouse	DF	correct implementation with readers priority	✓ verified
Shipping Warehouse	CR	WRONG implementation: missing <code>break</code> statement	✗ error detected: (oneMessageProcessed == false)
Shipping Warehouse	CR	WRONG implementation: <code>syncCond</code> is not consistent	✗ error detected: (wellFormedSyncCond == false)
Shipping Warehouse	CR	WRONG implementation: updates to <code>syncCond</code> are not complete enabling illegal invocations to proceed	✗ error detected: (cprePreservation == false)

Table 1 Summary of the results obtained when verifying several examples using KeY. Major attention is giving to detect which implementations have failures or bugs without considering the (finite and practical) time spent by the prover.

fixing the overall structure of the code (syntactic structure of the template) and the proof obligations that are verified using KeY. This helps with the traceability from the specification to the code.

Other applications of the Model-driven engineering in a large industrial context can be found in [4] that follows a similar approach to the one presented in this paper.

Apart from the KeY tool, several other verification tools such as Bogor [26], TACO [14] and Slam [5], were considered. All of them use model-checking as underlying verification technique. In principle, this limits the verification to concrete instances of the shared resources (i.e. those with a bound in the number of states of the corresponding transition system) and that is why we preferred to use a tool based on dynamic logic. However, in practice, tools based on model-checking can be a good alternative if formal verification is hard to achieve. We discarded the first two (due to the lack of support for concurrency) but tried Slam and, according to our (limited) experience, we must say it is a robust tool and the effort required to establish a number of key properties (e.g., *prop_safe_signal* and *prop_liveness*) was similar to the effort spent using KeY.

7 Conclusions and Future Work

We have presented a methodology for generating correct Java monitor implementations from high-level specifications of inter-process interaction (shared resource notation). The models formalize interaction by defining encapsulated state, an interface of *observationally atomic* actions and a state transition semantics. They can be presented in mathematical notation or as Java interfaces, using an extension of the JML specification language. Some features of our proposal include: (a) integration in a widely used programming language – the models are written as Java class interfaces annotated with a minimal extension of JML; (b) multilevel approach – some properties are established at model level, others are routinely ensured by the template syntax and, full adequacy of code to the template semantics is established by a formal verification tool, and (c) the integration with an existing state-of-the-art program verifier (KeY).

We have formalised several templates for code generation that allow good performance – i.e. without reevaluating synchronization conditions – when used with one implementation of priority (signal and continue) monitors. Two of

them (parameter- and thread-indexation) can deal with synchronization problems of moderate complexity, as shown in our running example or the use cases in [2].

For the case of distributed memory implementations using JCSP, we have also defined a similar set of templates. As their shared memory counterparts, they are based on a logical analysis of the formal specification and, roughly, the cases in which parameter indexation is preferred with monitors corresponds to the channel replication scheme and the cases where thread indexation is used correspond to the deferred request technique. In other words, some of the uses of condition queues are promoted to channels in the JCSP setting.

Although these templates could be used – under certain limitations on the datatypes involved – for fully automatic code generation, we have shown how a straightforward application of the templates can lead to inefficiencies, in particular, regarding the signaling code. However, the code manually optimized is prone to suffer from subtle bugs, even when the basic structure of the templates is preserved.

Our solution to the problem above is to associate proof obligations with certain points in the code template so as to ensure the scheme has been correctly implemented. These proof obligations are then *automatically* proved using the KeY program verification tool. We have conducted an experiment verifying correct and buggy implementations of the example presented using various degrees of optimization and we have been able to prove them right or wrong in KeY without human assistance. In the case of the buggy implementations the proof attempt fails and, furthermore, a counterexample of the proof can be generated that shows in which cases the specification will not hold. The full code used in the experiment and for other examples is available at our web site.

There are some remarkable issues in (the success of) our experiment. First, KeY is a tool for automatically verifying *sequential* code. Second, verification of concurrent software is in general hard to achieve due, in part, to the many interleavings that must be taken into account. Here, however, we have been able to verify some nontrivial safety and liveness properties without much complication. The crucial point here is that the code we are verifying is not *freeform* but follows a strict structure where the potential re-entry points in the method bodies are restricted. Second, the Java code has been instrumented to help KeY reason about these temporary interruptions of the sequential execution flow. In short, the tool works fine because the code implementing the shared resource is “almost” sequential.

Overall, one of the conclusions we can draw from our experiment is that the model based approach to the development of concurrent software also pays off when it comes to verify the correctness of implementations.

Currently, we are adding more examples to our collection. We are specially interested in a library of verified concurrent Java data structures, which will include, for the first time a formal specification of its synchronization behaviour as JML annotated Java interfaces.

Also, the lessons learned while encoding the proof obligations in KeY and the instrumentation techniques applied can be used as the starting point for an extension of KeY that natively supports shared resource specifications and priority monitors, thus introducing some high-level concurrency support in this great tool.

We are currently working on extending an existing JML compiler (OpenJML [12]) in order to support our interface annotations. This will enable advancing in two other directions. One is the mechanization of the logical analysis of the shared resource so that, for example, a tool can detect CPREs dependencies on parameter and advise on the right template to use. Second, automatic code generation for some subset of the shared notation specifications can be considered.

We have not shown examples of systems where processes interact through more than one shared resource. If that is the case, it suffices to apply the methodology to each resource, and that will ensure the correctness of each Java class implementation. System-wide correctness can be established either by verifying the state transition semantics of the composed system (work in progress) or by means of model-based testing (along the lines of [13]).

Acknowledgements

This research has been partially funded by Comunidad de Madrid grant S2013/ICE-2731 (*N-Greens Software*) and Spanish MINECO grant TIN2012-39391-C04-03 (*StrongSoft*).

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, H.P.: The key tool. *Software & Systems Modeling* **4**(1), 32–54 (2005). DOI 10.1007/s10270-004-0058-x. URL <http://dx.doi.org/10.1007/s10270-004-0058-x>
2. Alborodo, R.N.N., Mariño, J., Ángel Herranz: The Shared Resources home page. http://babel.upm.es/~rnnalborodo/sr_web/ (2014)
3. Araújo, J.E., Rebêlo, H., Lima, R., Mota, A., Kulesza, U., Sant’Anna, C.: An annotation-based approach for jscsp concurrent programming: A quantitative study. In: *Proceedings of the 1st Workshop on Modularity in Systems Software, MISS ’11*, pp. 7–11. ACM, New York, NY, USA (2011). DOI 10.1145/1960518.1960521. URL <http://doi.acm.org/10.1145/1960518.1960521>
4. Baker, P., Loh, S., Weil, F.: Model-Driven Engineering in a Large Industrial Context — Motorola Case Study, pp. 476–491. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). DOI 10.1007/11557432_36. URL http://dx.doi.org/10.1007/11557432_36

5. Ball, T., Cook, B., Levin, V., Rajamani, S., Ball, T.: Slam and static driver verifier: Technology transfer of formal methods inside microsoft. Tech. rep. (2004)
6. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS 4334. Springer (2007)
7. Brinch-Hansen, P.: Structured multiprogramming. Communications of the ACM **15**(7), 574–578 (1972). One of Brinch-Hansen’s seminal works on monitors.
8. Buhr, P.A., Fortier, M., Coffin, M.H.: Monitor classification. ACM Comput. Surv. **27**(1), 63–107 (1995). DOI 10.1145/214037.214100. URL <http://doi.acm.org/10.1145/214037.214100>
9. BV, V.S.T.: Dezyne home page. <http://www.verum.com/> (2015)
10. Carro, M., Herranz, Ángel., Mariño, J.: A model-driven approach to teaching concurrency. Trans. Comput. Educ. **13**(1), 5:1–5:19 (2013). DOI 10.1145/2414446.2414451. URL <http://doi.acm.org/10.1145/2414446.2414451>
11. Carro, M., Mariño, J., Ángel Herranz, Moreno-Navarro, J.J.: Teaching how to derive correct concurrent programs (from state-based specifications and code patterns). In: C. Dean, R. Boute (eds.) Teaching Formal Methods, CoLogNET/FME Symposium, TFM 2004, Ghent, Belgium, LNCS, vol. 3294, pp. 85–106. Springer (2004). ISBN 3-540-23611-2
12. Cok, D.R.: Openjml: Software verification for java 7 using jml, openjdk, and eclipse. In: Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014., pp. 79–92 (2014). DOI 10.4204/EPTCS.149.8. URL <http://dx.doi.org/10.4204/EPTCS.149.8>
13. Fredlund, L., Herranz-Nieva, Á., Alborodo, R.N.N., Mariño, J.: A testing-based approach to ensure the safety of shared resource concurrent systems (revised version of conference publication). Journal of Risk and Reliability (2015). Accepted for publication, to appear in 2016
14. Galeotti, J.P., Rosner, N., Lopez Pombo, C., Frias, M.: Taco: Analysis of invariants for efficient bounded verification. In: Tonella, Orso (eds.) International Symposium on Software Testing and Analysis (2010). URL <http://publicaciones.dc.uba.ar/Publicaciones/2010/GRLF10a>. Trento, Italy, July
15. Hamie, A.: Using Patterns to Map OCL Constraints to JML Specifications, pp. 35–48. Springer International Publishing, Cham (2015). DOI 10.1007/978-3-319-25156-1_3. URL http://dx.doi.org/10.1007/978-3-319-25156-1_3
16. Herranz, A., Mariño, J.: A verified implementation of priority monitors in Java. In: Proceedings of the 2011 international conference on Formal Verification of Object-Oriented Software, FoVeOOS’11, pp. 160–177. Springer-Verlag, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-31762-0_11. URL http://dx.doi.org/10.1007/978-3-642-31762-0_11
17. Herranz, A., Mariño, J., Carro, M., Moreno Navarro, J.J.: Modeling concurrent systems with shared resources. In: M. Alpuente, B. Cook, C. Joubert (eds.) Formal Methods for Industrial Critical Systems, Lecture Notes in Computer Science, vol. 5825, pp. 102–116. Springer Berlin Heidelberg (2009). DOI 10.1007/978-3-642-04570-7_9. URL http://dx.doi.org/10.1007/978-3-642-04570-7_9
18. Hilderink, G., Broenink, J., Vervoort, W., Bakkers, A.: Communicating java threads. In: Parallel programming and Java, Concurrent systems engineering series, vol. 50, pp. 48–76. IOS Press, Amsterdam, the Netherlands (1997). URL <http://doc.utwente.nl/16613/>
19. Hoare, C.A.R.: Monitors: an operating system structuring concept. Commun. ACM **17**(10), 549–557 (1974). DOI 10.1145/355620.361161. URL <http://doi.acm.org/10.1145/355620.361161>
20. Howard, J.H.: Signaling in monitors. In: Proceedings of the 2nd international conference on Software engineering, ICSE ’76, pp. 47–52. IEEE Computer Society Press, Los Alamitos, CA, USA (1976). URL <http://dl.acm.org/citation.cfm?id=800253.807647>
21. Klein, J., Levinson, H., Marchetti, J.: Model-driven engineering: Automatic code generation and beyond. Tech. Rep. CMU/SEI-2015-TN-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2015). URL <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=435414>
22. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. STTT **1**, 134–152 (1997)
23. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: H. Kilov, B. Rumpe, I. Simmonds (eds.) Behavioral Specifications of Businesses and Systems, The Springer International Series in Engineering and Computer Science, vol. 523, pp. 175–188. Springer US (1999). DOI 10.1007/978-1-4615-5229-1_12. URL http://dx.doi.org/10.1007/978-1-4615-5229-1_12
24. Mariño, J., Alborodo, R.N.N.: Communicating Process Architectures, chap. A Model-driven Methodology for Generating and Verifying CSP-based Java Code, pp. 85–108. Open Channel Publishing Ltd., Oxford, UK (2015). Accepted for publication, to appear in 2016
25. Pajic, M., Jiang, Z., Lee, I., Sokolsky, O., Mangharam, R.: From verification to implementation: A model translation tool and a pacemaker case study. In: 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium, Beijing, China, April 16–19, 2012, pp. 173–184 (2012). DOI 10.1109/RTAS.2012.25. URL <http://dx.doi.org/10.1109/RTAS.2012.25>
26. Robby, Dwyer, M.B., Hatcliff, J.: 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, pp. 267–276. ACM, New York, NY, USA (2003). DOI 10.1145/940071.940107. URL <http://doi.acm.org/10.1145/940071.940107>
27. Rodríguez, E., Dwyer, M., Flanagan, C., Hatcliff, J., Leavens, G., Robby: Extending JML for modular specification and verification of multi-threaded programs. In: A. Black (ed.) ECOOP 2005 - Object-Oriented Programming, Lecture Notes in Computer Science, vol. 3586, pp. 551–576. Springer Berlin Heidelberg (2005). DOI 10.1007/11531142_24. URL http://dx.doi.org/10.1007/11531142_24
28. Schaller, N.C., Hilderink, G.H., Welch, P.H.: Using Java for Parallel Computing: JCSP versus CTJ, a Comparison. In: P.H. Welch, A.W.P. Bakkers (eds.) Communicating Process Architectures 2000, pp. 205–226 (2000)
29. Welch, P.H., Austin, P.D., Brown, N.C.C.: Communicating Sequential Processes for Java (JCSP). <http://www.cs.kent.ac.uk/projects/ofa/JCSP/> (2012)

A fairSelect and Server loop instrumentation using synchronized methods

```

/* requires (syncCond.length==guards.length) &&
   @ (\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=0; 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;
}

```

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

```

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

while (chosenService!=-1){
  ... update syncCond array
  /* assert (\forall int j; 0<=j && j<MAX;
  @ syncCond[j]==CPREj);
  @ */
  chosenService=JCSPKey.fairSelect(syncCond,
    services);
  /* assume chosenService < guards.length &&
  @ chosenService >= -1 &&
  @ guards[chosenService].size() > 0 &&
  @ syncCond[chosenService];
  @ */
  switch(chosenService){
    case METHOD1:
      ...
      break;
    ...
  }
}

```

B WarehouseAccessMonitorNaive Instrumentation

Code corresponding to the instrumentation of the unblocking code of each operation belonging to the `WarehouseAccessMonitorNaive` class.

```

1 public class WarehouseAccessControlMonitorNaiveKeY{
2
3   /* ghost int awakenThreadC;
4   /* ghost int awakenThreadR;
5
6   /* public invariant N_WAREHOUSES > 0;
7   public static final int N_WAREHOUSES=2;
8   /* public invariant MAX_WEIGHT > 0;
9   public static final int MAX_WEIGHT=3;
10
11  // INNER STATE ATTRIBUTES
12
13  /* public invariant
14  @ corridor.length==N_WAREHOUSES - 1;
15  @ */
16  private /* spec_public */ boolean corridor[];
17
18  /* public invariant

```

```

19  @ weight.length==N_WAREHOUSES
20  @ && (\forall int i;
21  @ i >=0 && i < N_WAREHOUSES;
22  @ weight[i] >= 0 &&
23  @ weight[i] <= MAX_WEIGHT);
24  @ */
25  private /* spec_public */ int weight[];
26
27  // Monitor & conditions definition
28  /* public invariant
29  @ enteringWarehouse.length==N_WAREHOUSES &&
30  @ (\forall int i; i >=0 && i < N_WAREHOUSES;
31  @ enteringWarehouse[i].length==MAX_WEIGHT+1
32  @ && (\forall int j;
33  @ j >=0 && j <= MAX_WEIGHT;
34  @ enteringWarehouse[i][j] >= 0)
35  @ );
36  @ */
37  private /* spec_public */
38  int enteringWarehouse[][];
39
40  /* public invariant
41  @ (\forall int i; i >=0 && i < N_WAREHOUSES;
42  @ exitingWarehouse[i] >= 0) &&
43  @ exitingWarehouse.length==N_WAREHOUSES;
44  @ */
45  private /* spec_public */
46  int exitingWarehouse[];
47
48  /* public invariant signaled==0 || signaled==1;
49  private /* spec_public */ int signaled;
50
51  /* assignable enteringWarehouse[*], <-
52  exitingWarehouse[*];
53  /* assignable awakenThread, signaled;
54  /* diverges false;
55  /* prop_safe_signal
56  /* ensures
57  @ (\forall int j; j>=0 && j < N_WAREHOUSES;
58  @ (\forall int i; i>=0 && i < MAX_WEIGHT+1;
59  @ (enteringWarehouse[j][i]+1==\old(<-
60  enteringWarehouse[j][i] ==>
61  weight[j]+i <= MAX_WEIGHT)
62  @ )
63  @ )
64  @ &&
65  @ (\forall int i; i>=0 && i < N_WAREHOUSES;
66  @ (exitingWarehouse[i]+1==\old(<-
67  exitingWarehouse[i]
68  ==> ((i != N_WAREHOUSES-1 && <-
69  corridor[i]) || i==N_WAREHOUSES-1)
70  @ )
71  @ );
72  @ */
73  // prop_signal_0_1
74  /* ensures
75  @ (\forall int j; j>=0 && j < N_WAREHOUSES;
76  @ (\forall int i; i>=0 && i < MAX_WEIGHT +1; <-
77  i != N_WAREHOUSES - awakenThreadC ==>
78  (enteringWarehouse[j][i]==
79  \old(enteringWarehouse<-
80  [j][i]))
81  @ )
82  @ &&
83  @ (\forall int i; i>=0 && i < N_WAREHOUSES;
84  @ i != awakenThreadC ==> <-
85  exitingWarehouse[i]==\old(exitingWarehouse<-
86  [i]
87  @ )
88  @ )
89  @ &&
90  @ (awakenThreadC!=-1 ==> (
91  @ (awakenThreadC > N_WAREHOUSES ==>
92  @ (enteringWarehouse[N_WAREHOUSES - <-
93  awakenThreadC][awakenThreadR]+1
94  ==\old(enteringWarehouse)[<-
95  N_WAREHOUSES - awakenThreadC][awakenThreadR<-
96  ]))
97  @ ||
98  @ (awakenThreadC <= N_WAREHOUSES ==>

```

```

88     @      exitingWarehouse[awakenThreadC←
      ]+1== \old(exitingWarehouse)[awakenThreadC←
      ])
89     @      )
90     @      )
91     @      ;
92     @*/
93 // prop_signal_effective
94 /*@ ensures
95     @      signaled==1
96     @      ==>
97     @      (
98     @      ( awakenThreadC > N_WAREHOUSES ==>
99     @      enteringWarehouse[N_WAREHOUSES - ←
      awakenThreadC][awakenThreadR]+1 ==
100     @      \old(enteringWarehouse[N_WAREHOUSES←
      - awakenThreadC][awakenThreadR]))
101     @      &&
102     @      (awakenThreadC <= N_WAREHOUSES ==>
103     @      exitingWarehouse[awakenThreadC]+1== \old←
      (exitingWarehouse[awakenThreadC])
104     @      )
105     @      );
106     @*/
107 // prop_liveness
108 /*@ ensures
109     @      ( awakenThreadC!=-1 &&
110     @      ( awakenThreadC > N_WAREHOUSES ==>
111     @      (\old(enteringWarehouse[N_WAREHOUSES ←
      - awakenThreadC][awakenThreadR]) > 0
112     @      && weight[N_WAREHOUSES - ←
      awakenThreadC]+awakenThreadR <= MAX_WEIGHT
113     @      )
114     @      ) &&
115     @      (awakenThreadC <= N_WAREHOUSES ==>
116     @      (\old(exitingWarehouse)[awakenThreadC←
      ] > 0 &&
117     @      (
118     @      (awakenThreadC != N_WAREHOUSES-1 ←
      && corridor[awakenThreadC]) ||
119     @      (awakenThreadC==N_WAREHOUSES-1)
120     @      )
121     @      )
122     @      )
123     @      ) ==>
124     @      signaled==1
125     @      ;
126     @*/
127 private void dummyUnblockingCode(){
128 //@ set awakenThreadC=-1;
129 //@ set awakenThreadR=-1;
130 signaled=0;
131
132 /*@ loop_invariant
133     @      ((w >= 1 && w <= N_WAREHOUSES) || ←
      signaled==1) &&
134     @      (\forallall int l; l >= 1 && l < w ; ←
      enteringWarehouse[w][l]==0) &&
135     @      (awakenThreadC!=-1 ==> signaled==1)←
      ;
136     @ assignable enteringWarehouse[*];
137     @ decreases N_WAREHOUSES - w;
138     @*/
139 // for each warehouse...
140 for (int w=0; w < N_WAREHOUSES && signaled==0; ←
      w++) {
141 // and for each possible weight...
142 /*@ loop_invariant
143     @      ((j >= 1 && j <= MAX_WEIGHT+1) || ←
      signaled==1) &&
144     @      (\forallall int k; k >= 1 && k < j; ←
      enteringWarehouse[w][k]==0) &&
145     @      (awakenThreadC!=-1 ==> signaled←
      ==1);
146     @ assignable enteringWarehouse[*];
147     @ decreases MAX_WEIGHT+1 - j;
148     @*/
149 for (int j=0; j < MAX_WEIGHT+1 && signaled←
      ==0; j++) {
150 // we seek for an illegally blocked thread ←
      (0 could not be considered)
151 if ( weight[w]+j <= MAX_WEIGHT //cpreEnter

```

```

      && enteringWarehouse[w][j] > 0) {
152     enteringWarehouse[w][j]--;
153     //@ set awakenThreadC=N_WAREHOUSES+w;
154     //@ set awakenThreadR=j;
155     //@ assert weight[w]+j <= MAX_WEIGHT;
156     signaled++;
157 }
158 }
159 }
160 }
161 }
162 /*@ loop_invariant
163     @      ((i >= 1 && i <= N_WAREHOUSES) || ←
      signaled==1) &&
164     @      (\forallall int j; j >= 1 && j < i ; ←
      enteringWarehouse[i][j]==0) &&
165     @      (awakenThreadC!=-1 ==> signaled==1)←
      ;
166     @ assignable enteringWarehouse[*];
167     @ decreases N_WAREHOUSES - i;
168     @*/
169 // trying to unblock a robot that wants to ←
      leave the warehouse
170 // for each warehouse...
171 for (int i=0; i < N_WAREHOUSES && signaled==0; ←
      i++) {
172 // we seek for an illegally blocked robot ←
      that could exit a warehouse
173 if ( (i==N_WAREHOUSES-1 || corridor[i]) && ←
      exitingWarehouse[i] > 0) {
174     exitingWarehouse[i]--;
175     //@ set awakenThreadC=i;
176     //@ assert i==N_WAREHOUSES-1 || !corridor[i←
      ];
177     signaled++;
178 }
179 }
180 }
181 }

```

C Instrumented Unblocking Code of WarehouseAccessMonitorOpt

C.1 Unblocking code for exitWarehouse operation

```

178 //@ requires warehouse >= 0 && warehouse < ←
      N_WAREHOUSES;
179 //derived from POST
180 //@ requires warehouse != N_WAREHOUSES ==> ←
      corridor[warehouse];
181 //@ assignable enteringWarehouse[*];
182 //@ assignable exitingWarehouse[*];
183 //@ diverges false;
184 //prop_safe_signal
185 /*@ ensures
186     @      (\forallall int j; j>=0 && j < N_WAREHOUSES;
187     @      (\forallall int i; i>=0 && i < MAX_WEIGHT+1;
188     @      (enteringWarehouse[j][i]+1==\old(←
      enteringWarehouse)[j][i] ==>
189     @      weight[j]+i <= MAX_WEIGHT)
190     @      )
191     @      )
192     @      &&
193     @      (\forallall int i; i>=0 && i < N_WAREHOUSES;
194     @      (exitingWarehouse[i]+1==\old(←
      exitingWarehouse)[i]
195     @      ==> ((i != N_WAREHOUSES-1 && ←
      corridor[i]) || i==N_WAREHOUSES-1)
196     @      )
197     @      );
198     @*/
199 // prop_signal_0_1
200 /*@ ensures
201     @      (\forallall int j; j>=0 && j < N_WAREHOUSES;
202     @      (\forallall int i; i>=0 && i < MAX_WEIGHT +1;←
      i != N_WAREHOUSES - awakenThreadC ==>
203     @      (enteringWarehouse[j][i]==

```

```

204 @ \old(enteringWarehouse↔
205 @ )
206 @ &&
207 @ (\forall int i; i>=0 && i < N_WAREHOUSES;
208 @ i != awakenedThreadC ==> ↔
    exitingWarehouse[i]==\old(exitingWarehouse)↔
    [i])
209 @ )
210 @ )
211 @ &&
212 @ ( awakenedThreadC!=-1 ==> (
213 @ ( awakenedThreadC > N_WAREHOUSES ==>
214 @ (enteringWarehouse[N_WAREHOUSES - ↔
    awakenedThreadC][awakenedThreadR]+1
215 @ ==\old(enteringWarehouse)[↔
    N_WAREHOUSES - awakenedThreadC][awakenedThreadR↔
    ]))
216 @ ||
217 @ (awakenedThreadC <= N_WAREHOUSES ==>
218 @ exitingWarehouse[awakenedThreadC↔
    ]+1== \old(exitingWarehouse)[awakenedThreadC↔
    ])
219 @ )
220 @ )
221 @ ;
222 @*/
223 // prop_signal_effective
224 /*@ ensures
225 @ signaled==1
226 @ ==>
227 @ (
228 @ ( awakenedThreadC > N_WAREHOUSES ==>
229 @ enteringWarehouse[N_WAREHOUSES - ↔
    awakenedThreadC][awakenedThreadR]+1 ==
230 @ \old(enteringWarehouse[N_WAREHOUSES↔
    - awakenedThreadC][awakenedThreadR]))
231 @ &&
232 @ (awakenedThreadC <= N_WAREHOUSES ==>
233 @ exitingWarehouse[awakenedThreadC]+1== \old↔
    (exitingWarehouse[awakenedThreadC])
234 @ )
235 @ );
236 @*/
237 // prop_liveness
238 /*@ ensures
239 @ ( awakenedThreadC!=-1 &&
240 @ ( awakenedThreadC > N_WAREHOUSES ==>
241 @ (\old(enteringWarehouse[N_WAREHOUSES ↔
    - awakenedThreadC][awakenedThreadR]) > 0
242 @ && weight[N_WAREHOUSES - ↔
    awakenedThreadC]+awakenedThreadR <= MAX_WEIGHT
243 @ )
244 @ ) &&
245 @ (awakenedThreadC <= N_WAREHOUSES ==>
246 @ (\old(exitingWarehouse)[awakenedThreadC↔
    ] > 0 &&
247 @ (
248 @ (awakenedThreadC != N_WAREHOUSES-1 ↔
    && corridor[awakenedThreadC]) ||
249 @ (awakenedThreadC==N_WAREHOUSES-1)
250 @ )
251 @ )
252 @ ) ==>
253 @ signaled==1
254 @ ;
255 @ ;
256 @*/
257 private void exitWarehouseWnblockingCode(int ↔
    warehouse, int weight) {
258     signaled=0;
259     //@ set awakenedThreadC=-1;
260     // unblocking code
261     int wid=(warehouse==0)?0:warehouse-1;
262     int availableWeight=MAX_WEIGHT - weight[wid];
263     if (availableWeight > 0) {
264         /*@ loop_invariant
265         @ ((currentWeight >= 1 && currentWeight ↔
            <= availableWeight+1) || signaled==1) ↔
            &&

```

```

266 @
267 @
268 @
269 @
270 @
271 @
272 @
273 @
274 @
275 @
276 @
277 @
278 @
279 @
280 @
281 @
282 @
    (\forall int l; l >= 1 && l < ↔
    currentWeight ; enteringWarehouse[wid][↔
    l]==0) &&
    (awakenThreadC!=-1 ==> signaled↔
    ==1);
    @ assignable enteringWarehouse[wid][*];
    @ decreases availableWeight - currentWeight↔
    ;
    @*/
    for (int currentWeight=1; currentWeight <= ↔
    availableWeight && signaled==0; ↔
    currentWeight++){
        if (enteringWarehouse[wid][currentWeight] >↔
            0 ){
            enteringWarehouse[wid][currentWeight↔
            ]--;
            //@ set awakenedThreadC=wid;
            //@ set awakenedThreadR=currentWeight;
            //@ assert weight[wid]+currentWeight <=↔
            MAX_WEIGHT;
            signaled ++;
        }
    }
}

```

C.2 Unblocking code for enterWarehouse operation

```

62 /*@ requires warehouse >= 0 && warehouse < ↔
    N_WAREHOUSES;
63 //derived from POST
64 /*@ requires warehouse > 0 ==> !corridor[↔
    warehouse-1];
65 /*@ assignable enteringWarehouse[*];
66 /*@ assignable exitingWarehouse[*];
67 /*@ diverges false;
68 //prop_safe_signal
69 /*@ ensures
70 @ (\forall int j; j>=0 && j < N_WAREHOUSES;
71 @ (\forall int i; i>=0 && i < MAX_WEIGHT+1;
72 @ (enteringWarehouse[j][i]+1==\old(↔
    enteringWarehouse[j][i]) ==>
73 @ weight[j]+i <= MAX_WEIGHT)
74 @ )
75 @ )
76 @ &&
77 @ (\forall int i; i>=0 && i < N_WAREHOUSES;
78 @ (exitingWarehouse[i]+1==\old(↔
    exitingWarehouse[i])
79 @ ==> ((i != N_WAREHOUSES-1 &&↔
    corridor[i]) || i==N_WAREHOUSES-1)
80 @ )
81 @ );
82 @*/
83 // prop_signal_0_1
84 /*@ ensures
85 @ (\forall int j; j>=0 && j < N_WAREHOUSES;
86 @ (\forall int i; i>=0 && i < MAX_WEIGHT +1;↔
    i != N_WAREHOUSES - awakenedThreadC ==>
87 @ (enteringWarehouse[j][i]==
88 @ \old(enteringWarehouse↔
    [j][i]))
89 @ )
90 @ &&
91 @ (\forall int i; i>=0 && i < N_WAREHOUSES;
92 @ i != awakenedThreadC ==> ↔
    exitingWarehouse[i]==\old(exitingWarehouse)↔
    [i])
93 @ )
94 @ )
95 @ &&
96 @ ( awakenedThreadC!=-1 ==> (
97 @ ( awakenedThreadC > N_WAREHOUSES ==>
98 @ (enteringWarehouse[N_WAREHOUSES - ↔
    awakenedThreadC][awakenedThreadR]+1
99 @ ==\old(enteringWarehouse)[↔
    N_WAREHOUSES - awakenedThreadC][awakenedThread↔
    R]))
100 @ ||

```

```

101     @ (awakenThreadC <= N_WAREHOUSES ==>
102     @     exitingWarehouse[awakenThreadC<-
103         ]+1== \old(exitingWarehouse)[awakenThreadC<-
104         ])
105     @ )
106     @ ;
107     @*/
108 // prop_signal_effective
109 /*@ ensures
110 @ signaled==1
111 @ ==>
112 @ (
113 @     ( awakenThreadC > N_WAREHOUSES ==>
114 @         enteringWarehouse[N_WAREHOUSES - <-
115 @         awakenThreadC][awakenThreadR]+1 ==
116 @         \old(enteringWarehouse[N_WAREHOUSES<-
117 @         - awakenThreadC][awakenThreadR]))
118 @     &&
119 @     (awakenThreadC <= N_WAREHOUSES ==>
120 @         exitingWarehouse[awakenThreadC]+1== \old<-
121 @         (exitingWarehouse[awakenThreadC])
122 @     )
123 @ );
124 @*/
125 // prop_liveness
126 /*@ ensures
127 @ ( awakenThreadC!=-1 &&
128 @     ( awakenThreadC > N_WAREHOUSES ==>
129 @         (\old(enteringWarehouse[N_WAREHOUSES <-
130 @         - awakenThreadC][awakenThreadR]) > 0
131 @         && weight[N_WAREHOUSES - <-
132 @         awakenThreadC]+awakenThreadR <= MAX_WEIGHT
133 @     ) &&
134 @     (awakenThreadC <= N_WAREHOUSES ==>
135 @         (\old(exitingWarehouse)[awakenThreadC<-
136 @         ] > 0 &&
137 @         (
138 @             (awakenThreadC != N_WAREHOUSES-1 <-
139 @             && corridor[awakenThreadC]) ||
140 @             (awakenThreadC==N_WAREHOUSES-1)
141 @         )
142 @     ) ==>
143 @     signaled==1
144 @ );
145 @*/
146 private void enterWarehouseUnblockingCode(int <-
147     warehouse){
148     /*@ set awakenThreadC=-1;
149     signaled=0;
150
151     // unblocking code
152     // trying to wake up enterWarehouse calls
153     if (warehouse==0){
154         // only enterWarehouse 0 can be awakened
155         int availableWeight=MAX_WEIGHT - weight[<-
156         warehouse];
157         if (availableWeight > 0) {
158             /*@ loop_invariant
159             @ ((currentWeight>=1 && currentWeight <-
160             <= availableWeight+1) || signaled==1)<-
161             &&
162             @ (\forall int l; l >= 1 && l < <-
163             currentWeight ; enteringWarehouse[<-
164             warehouse][l]==0) &&
165             @ (awakenThreadC!=-1 ==> signaled<-
166             ==1);
167             @ assignable enteringWarehouse[warehouse<-
168             ][1..availableWeight];
169             @ decreases availableWeight - <-
170             currentWeight;
171             @*/
172             for (int currentWeight=1; currentWeight <= <-
173             availableWeight && signaled==0; <-
174             currentWeight++){
175                 if (enteringWarehouse[warehouse][<-
176                 currentWeight] > 0 ){
177                     enteringWarehouse[warehouse][<-
178                     currentWeight]--;
179
180                     /*@ assert weight[warehouse]+<-
181                     currentWeight <= MAX_WEIGHT;
182                     /*@ set awakenThreadC=N_WAREHOUSES+<-
183                     warehouse;
184                     /*@ set awakenThreadR=currentWeight;
185                     signaled++;
186                 }
187             } else { // warehouse > 0
188                 // exiting could perform iff a robot enter to<-
189                 any warehouse but the first
190                 if (exitingWarehouse[warehouse-1] > 0){
191                     exitingWarehouse[warehouse-1]--;
192                     /*@ set awakenThreadC=warehouse - 1;
193                     signaled ++;
194                     /*@ assert warehouse - 1==N_WAREHOUSES-1 || <-
195                     corridor[warehouse-1];
196                 }
197             }
198         }
199     }

```