


# A testing-based approach to ensure the safety of shared resource concurrent systems

Proc IMechE Part O:  
J Risk and Reliability  
2016, Vol. 230(5) 457–472  
© IMechE 2015  
Reprints and permissions:  
sagepub.co.uk/journalsPermissions.nav  
DOI: 10.1177/1748006X15614231  
pio.sagepub.com  


Lars-Åke Fredlund<sup>1</sup>, Julio Mariño<sup>1</sup>, Raúl NN Alborodo<sup>1,2</sup> and Ángel Herranz<sup>1</sup>

## Abstract

The paper describes a framework for testing a class of safety-critical concurrent systems implemented using shared resource specifications. Shared resources contain declarative specifications of process interaction that can be used to derive, in a *model-driven* way, the most critical parts of a concurrent system. Here, we propose their use to build a state-based model that will help in testing a *real* implementation of the resource. The framework has been implemented using Erlang and QuickCheck and its source code is available. The paper also provides a novel parametric operational semantics for shared resources with scheduling policy annotations and a methodology to guide test-case generation from the shared resource specifications and a classification of common mistakes. We illustrate our framework by applying it to testing Java implementations of a prototypical automated shipping plant.

## Keywords

Testing, concurrency, automated, model-based, property-based, black box, shared resources, Java, Erlang, QuickCheck

Date received: ; accepted:

## 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 do not provide the same evidence for checking if the system works properly as in the case of sequential software. To make things worse, the development of concurrent software is affected by shortcomings in programming language and library support, design methodologies, the availability of properly trained practitioners, and so on.

A typical case where the choice of an unsuitable programming language can make things worse is that of Java. Programming safety-critical applications in Java is tempting (except if the targeted system has hard real-time constraints due to e.g. the presence of automatic garbage collection) since there is a large body of Java programmers available. However, the language and its libraries provide a large number of different concurrency primitives, but their limitations are often not well understood. Moreover, these concurrency primitives are generally low-level constructs, primarily targeting *efficient*

execution rather than *safe* execution, thus constituting poor choices for implementing safety-critical systems.

In this work we attempt to improve the situation in two ways. First, we use a particular formalism for the specification of a class of shared resources<sup>1</sup> (in this paper henceforth simply called *shared resources*) to provide a high-level specification of (discrete-time) process interaction. Secondly, we present a framework for performing property-based testing on the code implementing that resource. This is accomplished by creating a finite-state machine (FSM) from the specification, which is used to automatically generate test cases to be run under Quviq QuickCheck.<sup>2</sup>

## Shared resources

Our approach to concurrent software design distinguishes active entities (threads, processes) from passive

<sup>1</sup>Babel Group, Universidad Politécnica de Madrid, Spain

<sup>2</sup>IMDEA Software Institute, Madrid, Spain

## Corresponding author:

Julio Mariño, Escuela Técnica Superior de Ingenieros Informáticos UPM, Campus de Montegancedo s/n, 28660 Boadilla del Monte, Spain.  
Email: jmarino@fi.upm.es

ones (resources, shared memory locations). The latter represent all kinds of interactions among the former, and are formally modelled using an abstraction (shared resource specifications) that contains a clear interface (which can be invoked from processes) and a transactional transition semantics.

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

The context of this work is a research line in which we advocate the use of the *model-driven* approach for the development of correct concurrent software. In this paradigm, code is not written directly from (informal) requirements specifications, but produced from some intermediate (formal) representation of those requirements: the *model*. Of course, introducing an extra abstraction and its accompanying burden of formality can increase certain costs and make a steeper learning curve, but that is compensated for by an improvement in the quality of the development process and the resulting software. Although the terms ‘model-driven’ and ‘model-based’ are relatively recent, the concepts have been in use for several decades, for example compiler construction from grammar, databases from entity/relationship models, and so on.

We summarize some of the benefits of the model-driven approach (both in general and for the specific case of concurrent software).

1. Formalizing (part of) the requirements reduces ambiguity in the problem statement.
2. Formal models can be the subject of experiments aimed at *early requirement validation*. That is, a mathematical model can be formally verified for detecting inconsistencies or other flaws. In Herranz et al.<sup>1</sup> shared resources are translated into TLA (<http://www.research.microsoft.com/users/lamport/tla/tla.html>) for early detection of deadlocks, starvation, and so on. This prevents these issues from reappearing at the code generation stage.
3. Code is not written from scratch but is *generated* or *distilled* (semiautomatically) from the model. This brings several benefits. One of them is *portability*. This is especially relevant for concurrent software production, given the volatility of certain languages. A second benefit is robustness against changes in the requirements: modifying concurrent code by hand may introduce more errors than re-generating it. Finally, the generative approach may reduce production costs at this stage. In Carro et al.<sup>3</sup> and Mariño et al.<sup>4</sup>

pattern-based code generation is described for Ada and Java.

4. Models can help in the validation, verification and test-case generation of the code obtained from the previous phases.

This work is our first contribution to the last point above. We assume that our code has been generated from patterns that enforce some discipline in the use of error-prone concurrency primitives, thus alleviating some of the aforementioned language-related issues. However, as the experiment in Section 6 shows, even if the critical code has been structured according to a shared resource specification, average programmers can still make mistakes that often have to do with subtleties of the target language synchronization primitives. Clearly we must at least test, systematically, the resulting implementation.

### Property-based testing of shared resources

The second component of the methodology is thus the extensive use of property-based testing to (i) use a small finite-state model to automatically generate, semi-randomly, intelligible test cases, and (ii) automatically decide whether test execution is successful (using the declarative specification as a base). The testing tool we use, Quviq QuickCheck,<sup>2</sup> a variant of the well-known QuickCheck tool,<sup>5</sup> has excellent support for testing stateful code. Essentially we build a model of the system being tested (the shared resource), and use the model both to derive tests and to judge the correctness of the execution of the system being tested by comparing it with the execution of the system model.

Although tests are derived in part randomly, this does not mean that we miss the possibility of deriving good test suites with a desired coverage according to some testing coverage measure. We can semi-randomly search for a set of test cases that fulfil certain coverage criteria, for example regarding states and transitions in the model machine which represent a subset of the shared resource transition semantics. Indeed, there is experimental evidence that suggests that combining the two approaches (random testing and coverage-directed test-case generation) yields better results than applying any of them in isolation.<sup>6</sup>

To evaluate the effectiveness of the approach, we have applied it to the task of specifying and verifying a prototypical concurrent safety-critical system, a warehouse complex where autonomous robots move around fulfilling shipping orders. The term ‘safety’ is used throughout the paper with the following meaning. We will say that some implementation of a shared resource is *unsafe* when it allows the execution of a resource operation from a state where some of its preconditions do not hold, or conversely, where executing some operation leaves the resource in a state incompatible with the operation’s postcondition or the resource invariant. (As will be explained in the next section, we consider

two kinds of preconditions. Invoking an operation in a state that violates the *synchronization* precondition, denoted CPRE, blocks the calling thread (until the CPRE holds). Traditional preconditions, denoted pre, should be treated as exceptions when violated.) Sometimes, we will also generate tests aimed at detecting *liveness* issues: mostly the case where some threads invoking operations on a resource are blocked unnecessarily (lack of concurrency), cannot be given an upper bound on the number of resource operations served before being given access to the resource once the preconditions are met (starvation), or some priority requirement is not met (largest job first, first-come-first-served, etc.) The specific safety-critical aspect we will consider in the case study is that the weight of the robots, in any given warehouse, should never exceed a certain maximum weight. To evaluate the usefulness of automatic testing, we proceeded to test a large (around 100) number of implementations of the warehouse control system provided by undergraduate students at the Technical University of Madrid. A prototype implementation of the testing framework is available as an open-source project (at [git@bitbucket.org:fredlund1/shared\\_resources\\_erlang.git](https://github.com/fredlund1/shared_resources_erlang.git)). To test a shared resource using this testing framework three components are needed: (i) a model implementation of the resource in the Erlang programming language, (ii) a definition in Erlang of the scheduling policy to use during testing (several standard policies are pre-defined), and (iii) a QuickCheck state machine that decides which calls to issue to the shared resource being tested.

### Related work

Model-based testing<sup>7</sup> is currently a very active research area. A number of recent (mainly theoretical) proposals define frameworks for model-based testing of concurrent systems<sup>8,9,10</sup> based on previous work on testing labelled transition systems.<sup>11,12</sup> Our proposal tries to be fundamentally practical, and, although the underlying behaviour of a shared resource can be ultimately expressed in terms of a labelled state transition system, the idea is to express tests at the same level of abstraction as the specifications, that is, coarse-grained resource operations, pre- and postconditions, resource invariants, and so on.

In Betin-Can et al.<sup>13</sup> and Yavuz-Kahveci and Bultan,<sup>14</sup> techniques for verifying concurrent controllers are presented by distinguishing two components: concurrent controllers and their clients. Although the approach is similar to ours, their works present verification over the specification of the concurrent controller component (very much like the use of TLA by Herranz et al.<sup>1</sup>) whilst our present proposal discusses black-box testing on a given Java implementation. Moreover, client interleaving is not tested with the same level of detail as in our proposal.

UPPAAL Tron<sup>15</sup> seems to be a good proposal for model-based testing over timed FSMs. One limitation

of the tool is the need to (explicitly) describe all reachable states beforehand in order to express the behaviour of a given shared resource. We think that our shared resources are in general more concise for (implicitly) expressing a complex FSM for multiple process interaction.

The technique proposed by Cheung and Chow,<sup>16</sup> while similar in spirit, falls more on the side of early design validation.

The article is organized as follows. First, in Section 2 we introduce the warehouse case study which is used as a running example throughout the article. Then, in Section 3, we introduce the formalism for defining shared resources and provide an operational semantics describing their exact behaviour.

In Section 4 we describe how shared-resources-based safety-critical systems are tested using our approach. In Section 5 we introduce the QuickCheck property-based testing tool, which is the test used in the testing framework. The testing framework is evaluated by applying it to a number of implementations of the warehouse control system in Section 6. Finally, Section 7 draws conclusions from the work realized so far, and details issues for future work.

### Case study

The running example used in the article is the control system for a warehouse complex serviced by a set of autonomous robots. An example warehouse complex, with robots, is depicted in Figure 1.

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, and so on, 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 monotonically 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_IN_WAREHOUSE` when a new robot enters the warehouse. It is permitted for the total weight in a warehouse to be 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 space for a single robot.

In Figure 1, the constant `MAX_WEIGHT_IN_WAREHOUSE` is set to 1000 kg, and thus, for example, we can see that since the total weight in warehouse 0 is  $500 + 200 + 200 = 900$  the robots with weights 200 and 300 that want to enter should be blocked, while the robot with weight 100 can be permitted to enter. Moreover, as the corridor between warehouses 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.

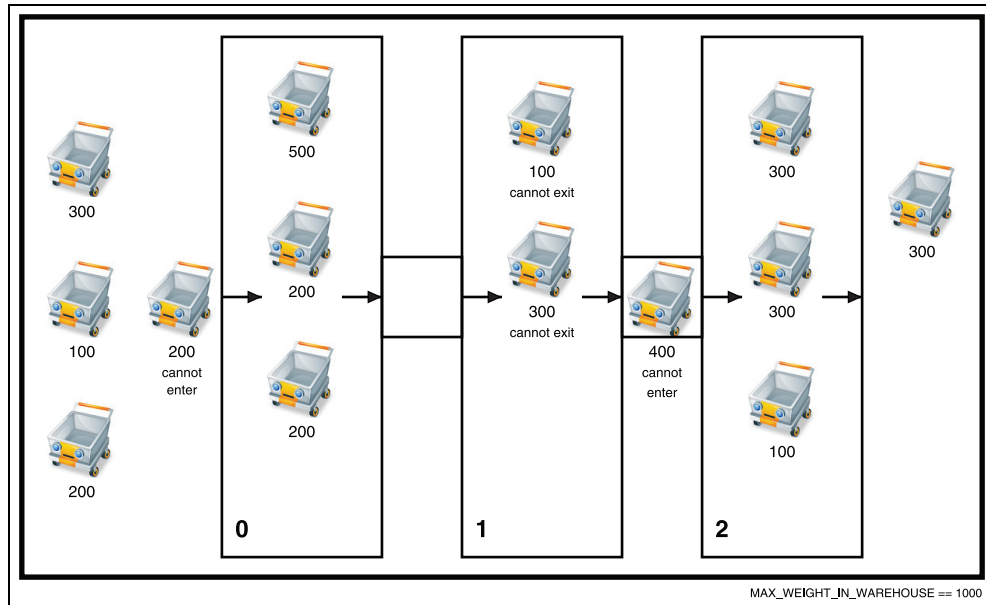


Figure 1. Warehouses and robot movements.

## Shared resources

One commonly used mechanism for controlling interactions between concurrent processes is to impose some form of central control, to serialize potentially conflicting requests. The *shared resources* introduced by Herranz et al.<sup>1</sup> are one such centralized mechanism. A shared resource has a state, and defines a set of methods (or operations) which provide the only mechanisms to modify this resource state. Such methods are guaranteed to execute atomically, at least from an observational point of view. This means that the concurrent execution of two calls to any method of a certain resource is either forbidden (by mutual exclusion) or, if allowed, the observed behaviour of any concurrent set of calls is equivalent to some sequential execution (linearizability<sup>17</sup>).

## Safety specification

The behaviour of a method is described declaratively, using a set of rules. A method *precondition* (PRE) describes requirements upon the caller of a method, and the method *postcondition* (POST) details the effect of the execution of a method on the state of the shared resource. Characterizing method behaviour using preconditions and postconditions is rather standard in non-concurrent formalisms; to cater for concurrency the shared resources formalism introduces a new condition, the *concurrency precondition* (CPRE) which describes *when* a method can be executed, considering the state of the shared resource.

As a simple introductory example, consider a typical *readers & writers* system.<sup>18</sup> Reader processes request permission to read a shared document by invoking

*beforeRead*. On return of that operation, which may be blocking, the process may access the document and, when done, it indicates this by invoking *afterRead*. Analogously, writer processes perform the sequence consisting in invoking *beforeWrite*, modifying the document, and invoking *afterWrite*. The formal specification is shown in Figure 2. We have decided to represent the internal state as a pair of naturals, but other representations (e.g. using a Boolean for writers) are possible. Observe also how the ‘before’ methods can block the invoking process ( $CPRE \neq true$ ) while the ‘after’ methods are non-blocking. Under the assumption that processes can only interact via the shared resource, communication takes place by changes in the resource’s internal state while synchronization happens only when a process invokes a method whose CPRE does not hold in a certain state. Observe that the state prior to a method call is referred to in the post clauses using references of the form  $x^{in}$ . Finally, we assume processes to respect the ‘before; access; after’ protocol, in other words, no process can invoke an after method without having invoked the corresponding before method. That would lead to a violation of the resource invariant. This behaviour is summarized graphically in Figure 3, which represents the transition system generated by the formal specification. Note how the initial state is the only one that allows for both ‘before’ transitions to happen.

Getting back to our case study, Figure 4 contains the specification of the safety control part. 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  $w$ ;

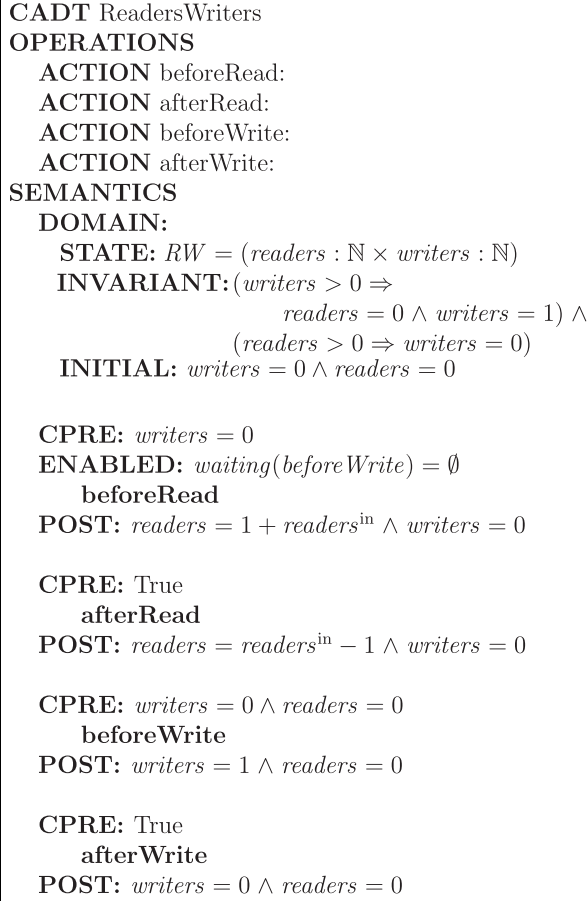


Figure 2. Formal specification of the readers &amp; writers resource.

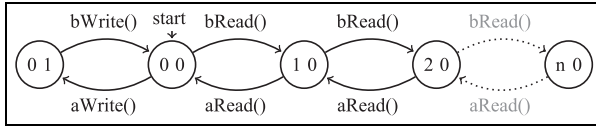


Figure 3. Transition system generated by the readers &amp; writers shared resource. The first component of the internal state indicates the number of active readers whilst the second one represents the active writers.

- *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 map from a warehouse to weight (a natural number), and *occupied*, a map from a warehouse to a Boolean. Intuitively, *weight* should correspond to the accumulated weight in the warehouse, and *occupied* ( $n$ ) is true if there is a robot present in corridor  $n$  leading from the warehouse.

Initially, as specified in the INITIAL clause, the weight in all warehouses is zero, and no robot is present in any corridor. The resource has an invariant over the state, as specified by the INVARIANT clause, that is, that the weight in a warehouse should always be less

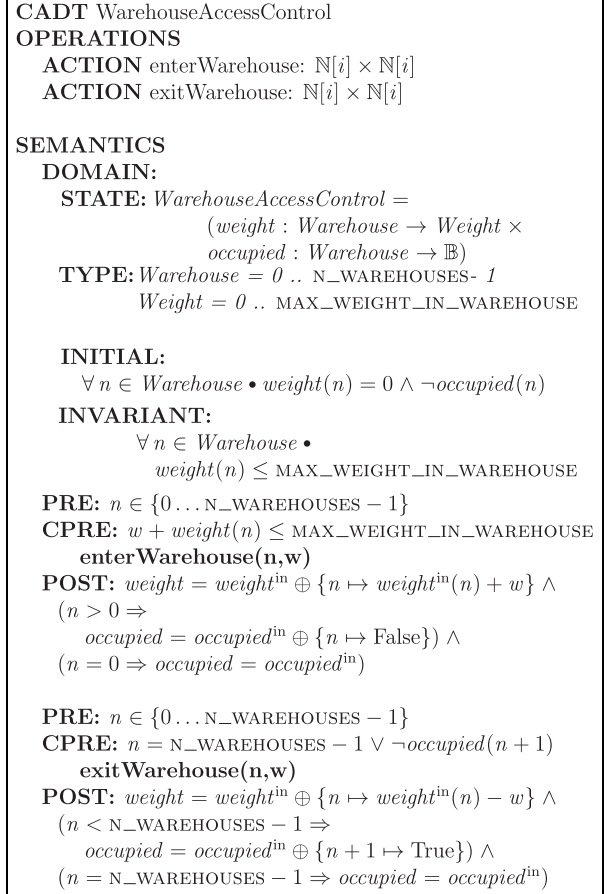


Figure 4. Specification of the robot controller.

than or equal to the maximum weight  $MAX\_WEIGHT\_IN\_WAREHOUSE$ .

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 when access is safe, that is, 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 of an operation that every call must satisfy.

Similarly, a robot should always call the operation *exitWarehouse(n, w)* to ask for permission to leave a warehouse. The CPRE ensures that the call does not return until the corridor leading away from the warehouse is free of robots. A restriction on the caller to these operations is that the weight  $w$  provided as an argument of the operation *exitWarehouse(n, w)* when asking for permission to leave a warehouse must be identical to the weight provided when asking for permission to enter the warehouse, that is,

*enterWarehouse*( $n, w$ ). In other words, 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*, for example *enterWarehouse*( $n + 1, w + \text{cargoWeight}$ ).

Examples of shared resource specifications of varying complexity can be found at [http://babel.upm.es/~rnnalborodo/sr\\_web/](http://babel.upm.es/~rnnalborodo/sr_web/).

### Scheduling specification

As a major new contribution in this article we further extend the declarative specification of a shared resource to permit the specification of scheduling policies, that is, to decide given multiple competing calls whose concurrency preconditions are true, and which call should be executed first. As our goal is to enable automatic testing of implementations of shared resources, having a formal and precise definition of such scheduling policies is crucial.

Thus, in this article, a shared resource specification consists of two parts: a *safety* specification which provides the pre-, post-, and concurrency preconditions (this is described above), and a *scheduling* specification which, for each method of the operation, defines constraints on the scheduling of calls to the method. Such constraints are specified in three auxiliary clauses that deal with the new *scheduling state* of the resource. These are the following.

ENABLED describes when a call may be scheduled, given the call, the scheduling control information associated with the call, and the scheduling state. By default, we assume the scheduling state to contain an ordered sequence of pending calls (what we call a *history*) and a number of predefined functions to access histories that match a certain pattern. The readers & writers specification in Figure 2 shows an example of this. The ENABLED clause specifies the additional requirement of no pending call to *beforeWrite* in order to serve a call to *beforeRead*. This is a (maybe drastic) way of dealing with the risk of writer starvation. For finer control over scheduling states, two other clauses are available: WAITING, which modifies the scheduling state right after a call (whose precondition is true) is invoked; and POST\_WAITING, which describes possible effects on the scheduling state right after executing a call.

In our robots example, there are two obvious variations on scheduling: to impose a first-in/first-out (FIFO) discipline in the advance of robots (which could reduce the overall throughput of the system), or to give preference to robots with the heaviest loads (at the risk of lighter robots starving). Assuming the existence of a function *older* on histories such that *older(pat)* returns the subhistory of pending calls matching *pat* with an arrival time older than the invoking one, the FIFO strategy can be expressed as

**PRE:**  $n \in \{0 \dots N\_WAREHOUSES - 1\}$   
**CPRE:**  $w + \text{weight}(n) \leq \text{MAX\_WEIGHT\_IN\_WAREHOUSE}$   
**ENABLED:**  $\text{older}(\text{enterWarehouse}(n, \_)) = \emptyset$   
*enterWarehouse*( $n, w$ ) ...  
**POST:** ...

and the *largest robot first* policy, as

**PRE:**  $n \in \{0 \dots N\_WAREHOUSES - 1\}$   
**CPRE:**  $w + \text{weight}(n) \leq \text{MAX\_WEIGHT\_IN\_WAREHOUSE}$   
**ENABLED:**  $\forall \text{enterWarehouse}(n, w') \in$   
 $\text{waiting}(\text{enterWarehouse}(n, \_)) \bullet w' \leq w$   
*enterWarehouse*( $n, w$ ) ...  
**POST:** ...

The machinery needed to implement histories, scheduling states and the helper functions is described in the following subsection.

### Resource semantics

In this section an operational semantics is provided for shared resources. In order to make it simpler, we do not model the notion of external processes which are the clients of the given resource.

Let the configuration of the shared resource be a tuple  $\langle in, calls, sa, ss, out \rangle$  where:

*in* is the ‘channel’ upon which calls made to the shared resource arrive, a sequence of calls in arrival order. The delivery mechanism for calls issued to a shared resource is left to the implementation. A call is a pair consisting of a shared resource operation name and a list of parameters:

$$\text{methodCall} \triangleq \mathcal{P}(\text{Operations} \times \tau)$$

$$in \triangleq \mathcal{P}(\text{methodCall})$$

*calls* is a multiset of calls with true preconditions that have not yet completed associated with additional scheduling control information generated by the WAITING clause:

$$\text{calls} \triangleq \mathcal{P}(\text{cinfo} \times \text{methodCall})$$

Notice that *cinfo* could be of any type strictly attached to the scheduling component provided by the tester. For instance, to represent a FIFO policy, we can consider  $\text{cinfo} \equiv \mathcal{N}$ .

*sa* is the internal state of the shared resource (initially as defined by INITIAL clause).

*ss* is the schedule component (initial state may vary depending on the policy adopted). Again, the exact implementation mechanism for delivering such a continuation permission is left to the implementation scheduling state.

*out* is a multiset used for denoting a decision by the shared resource to permit a caller of a method to continue its execution once the call has terminated:

$$\text{out} \triangleq \mathcal{P}(\text{calls} \times \text{status} \times \text{returnValue})$$



PRE	:	$methodCall \rightarrow \mathbb{B}$
WAITING	:	$(methodCall \times schedulingState \times resourceState) \rightarrow schedulingInfo \times schedulingState$
ENABLED	:	$(methodCall \times schedulingInfo \times schedulingState \times resourceState) \rightarrow \mathbb{B}$
POST	:	$(methodCall \times resourceState) \rightarrow resourceState$
POST_WAITING	:	$(methodCall \times schedulingInfo \times schedulingState \times resourceState) \rightarrow schedulingState$
$\neg PRE(call)$		
$\frac{}{\langle call \cdot in, calls, sa, ss, out \rangle \rightarrow \langle in, calls, sa, ss, out \rangle} \text{ (REJECT)}$		
$\frac{PRE(call) \quad \langle cinfo, ss' \rangle = WAITING(call, ss, sa)}{\langle call \cdot in, calls, sa, ss, out \rangle \rightarrow \langle in, calls \cup \{ \langle cinfo, call \rangle \}, sa, ss', out \rangle} \text{ (CHECK-IN)}$		
$\frac{\langle cinfo, call \rangle \in calls \quad CPRE(call, sa) \quad ENABLED(call, cinfo, ss, sa) \quad \begin{array}{l} calls' = call \setminus \{ \langle cinfo, call \rangle \} \\ sa' = POST(call, sa) \\ ss' = POST\_WAITING(call, cinfo, ss, sa) \end{array}}{\langle in, calls, sa, ss, out \rangle \rightarrow \langle in, calls', sa', ss', out \cdot call \rangle} \text{ (EXECUTE)}$		

**Figure 5.** Operational semantics for shared resources with scheduling.

$$status \triangleq \{failed, succeed\}$$

Note that in this article, for reasons of clarity of presentation, we abstract away from the return values of calls; the extension to the shared resource specification, and the operational semantics, to permit a call to return a value is trivial. The operational semantics rules, then, are defined on the following format:

$$\frac{c_1 \dots c_n}{\langle in, calls, sa, ss, out \rangle \rightarrow \langle in', calls', sa', ss', out' \rangle}$$

indicating that a shared resource specification in the state  $\langle in, calls, sa, ss, out \rangle$  can transit into state  $\langle in', calls', sa', ss', out' \rangle$  provided that conditions  $c_1, \dots, c_n$  hold. The semantic rules are presented in Figure 5. The semantics is parametric over the concrete definition of five instrumental functions used to manipulate the scheduling state at different stages of a call being processed.

The REJECT rule triggers when a call does not satisfy its precondition. If the call does satisfy the precondition (CHECK-IN rule) the scheduling state is updated and the call together with ‘call information’ data is stored in the set of calls. The third and final rule (EXECUTE) models the execution of a method call. Any received call, which is enabled (i.e. CPRE holds) and which is enabled to execute according to the scheduler (i.e. ENABLED holds) can be executed. The result of the execution is that both the safety state is updated (through the POST rule) and the scheduling state is updated (through the POST\_WAITING rule), and the call is added to the output (signalling that the caller may continue).

A run from a state  $s_0$  over a shared resource is a finite sequence of states  $s_0, s_1, \dots, s_n$  such that  $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n$ . Denote by  $runs(s_0)$  the (finite) set of runs from state  $s_0$ , and by  $finals(s_0)$  the set  $\{s_i | s_i \in runs(s_0)\}$ , that is, the set of final states of

runs, and by  $completed(s_0)$  the set of states  $\{s' | s' \in finals(s_0) \wedge \neg \exists s''. s' \rightarrow s''\}$  corresponding to final states in runs which cannot be extended.

Moreover, the implementation of a resource may be progressing; in the absence of further calls such an implementation will eventually complete a call whose concurrency precondition and scheduling condition are continuously true.

For example, the ‘writers over readers’ priority scheduling policy

**CPRE:**  $writers = 0$

**ENABLED:**  $waiting(beforeWrite) = \emptyset$

**beforeRead**

**POST:**  $readers = 1 + readers^{in} \wedge writers = 0$

can be formalized as

$$\begin{aligned} schedulingState &\triangleq \mathbb{N} \\ PRE(beforeRead) &\triangleq true \\ PRE(beforeRead) &\triangleq true \\ WAITING(beforeRead, ss, sa) &\triangleq \langle \rangle, ss \\ WAITING(beforeWrite, ss, sa) &\triangleq \langle \rangle, ss + 1 \\ ENABLED(beforeRead, \_, ss, sa) &\triangleq ss = 0 \\ ENABLED(beforeWrite, \_, ss, sa) &\triangleq ss \\ POST\_WAITING(beforeRead, cinfo, ss, sa) &\triangleq ss \\ POST\_WAITING(beforeWrite, cinfo, ss, sa) &\triangleq ss - 1 \end{aligned}$$

## Testing resources

There are different aspects of a system implemented using shared resources that we can test. We can for instance focus on testing the specification itself, to *validate* that the specification is internally consistent, and that it faithfully expresses the informal requirements an implemented system should satisfy. An example of a consistency property is that all postconditions should preserve the resource invariant.

Here, instead, we mainly focus on the task of *verifying* that an implemented system faithfully conforms to the resource specification on which it is based. Apart

from testing the safety (CPRE, POST) aspects of the system being tested, there are often implementation requirements on the order in which the implemented system services the calls whose concurrency preconditions hold.

For the warehouse example, for example, there is just a single requirement on servicing calls, to enforce progress:

*If the set of calls with true concurrency preconditions is non-empty, the system must eventually select a call to execute.*

We can illustrate the semantics of this requirement by an example, assuming that the maximum weight permitted in warehouse 0 is 1000 kg:

```
enterWarehouse(0,900)
enterWarehouse(0,200)
enterWarehouse(0,100)
```

We assume that calls are made sequentially. The first call does not block, as  $900 \leq 1000$ . The second call blocks, as  $900 + 200 > 1000$ . The third call is permitted by the concurrency precondition as  $900 + 100 \leq 1000$ , and thus cannot be blocked for infinitely long. Such requirements are expressed formally by associating a scheduling specification with the system being tested.

We will test a shared resource by developing a model for the behaviour of the resource as a Quviq QuickCheck<sup>2</sup> state machine. In the following we assume that the system is implemented using Java, although this is not crucial to the approach.

The first question to ask is what errors we can expect programmers to make when implementing a shared resource. The errors can be broadly separated into three classes.

**non-atomic:** calls are not evaluated atomically. That is, the evaluation of the concurrency preconditions and postconditions (or of scheduling checks) of different calls are interleaved, although the concurrency precondition and postcondition of a given call should be evaluated in sequence. These errors are likely due to basic misunderstandings with regards to using a particular concurrency feature in the implementation language. To find such errors concurrent calls to the shared resource must be issued.

**bad-spec-impl:** badly implemented CPREs or POSTs. To detect such errors issuing a sequence of sequential calls is sufficient.

**bad-sched:** bad implementation of scheduling requirements. That is, the programmer has made mistakes in the selection of a call candidate eligible to enter the resource; this can be a difficult task due to ordering constraints and the manner in which blocked tasks must be woken up. Correctly programming this functionality in for example Java is not an easy task,<sup>4</sup> and we can expect to see many errors here.

To detect such errors we must be able to observe which pending calls were unblocked by the execution of a non-blocking call. That is, if the concurrency preconditions for all pending calls in a shared resource are false, and a new call  $call_1$  arrives whose precondition is true, we should observe which pending calls  $call_2, \dots, call_n$  are unblocked due to the execution of  $call_1$ .

### Observational power of tests

An implementation of a shared resource will be tested by issuing, concurrently, a number of calls to the resource and observing which calls complete (and which remain blocked).

With regards to the semantics presented in Section 3.2, this corresponds to providing a sequence of calls as input (in the *in* component), and observing what calls complete, and in what order (i.e. noting the contents of the *out* component).

In the following it is assumed that implementations of shared resources cannot be instrumented in any way: remember this is black-box testing. The only interactions possible with such an implementation is to create a new resource, and issue calls to it and observe whether a call completes.

Concretely, if calls are made concurrently, there is no possibility of observing either in which order calls *arrive* at the shared resource (i.e. the ordering of calls in *in*), or the order in which calls complete (i.e. the ordering of calls in *out*). Moreover, there is no fail-safe method to observe when a call is blocked, for example because its concurrency precondition is false, or because it is not permitted to complete due to scheduling constraints, as we cannot distinguish a very slow but non-blocked call (e.g. because the computation of the postcondition is time-consuming) from a truly blocked call.

To observe whether a call blocks then, a pragmatic approach is taken. A test is separated into a number of *test phases*, where the exact number of phases is chosen randomly. Each phase starts by issuing a number of concurrent calls to the shared resource. Then the testing code waits a sufficiently long time for all calls to complete, and calls which have not completed within the time limit are considered blocked. Then a new test phase begins, with issuing new concurrent calls to the resource, potentially unblocking old blocked calls too. Clearly this approach is unsound in general, if the completion of a call can take a unknown amount of time. However, for most systems there are requirements on the maximum amount of time spent on calls, as the input/output (I/O) and time-consuming operations will be part of the client processes code, while the code implementing resources is assumed to be essentially *pure* and *instantaneous*. In practice the time spent waiting for calls to complete in a phase is a test parameter, which can be specified at the start of testing. Currently, all test phases use the same waiting time.



As a conclusion, if we abstract away from the return values of calls (as is done in this article for presentation purposes), the test code can judge the success of a test solely based on which calls complete, and which calls block during each test phase.

### Implementing tests

When testing a shared resource there are two essential tasks: (i) deciding which sequences of calls to issue to the shared resource, and (ii) deciding whether the execution of a set of calls was successful or not.

**Deciding on the contents of a test.** Although it would be possible to generate purely random calls to a shared resource undergoing testing this is often not desirable, as most such randomly generated calls would likely be rejected immediately by a call precondition, or would simply be nonsensical, and moreover, the test *coverage* (with regards to some test measure such as e.g. line coverage, state coverage, or modified condition/decision coverage (MC/DC).) would likely be very poor. Instead of generating purely random calls we use a QuickCheck state machine to generate sensible calls, which are tailored to the particular shared resource under test. That is, the responsibility of dealing with the inherent incompleteness of test coverage is transferred (in part) to the test designer. As explained earlier, a test case is composed of a number of test phases. The QuickCheck state machine for call generation issues in each test phase either a single call, or a small number of concurrent calls. As an example, Section 6.1 discusses how calls are generated for the robot case study.

**Deciding on a test result.** The decision of whether the execution of a test was successful is broken down into one decision for each test phase.

In the following we will consider progressing implementations only, in other words, implementations that eventually execute a continuously enabled call, and moreover, we assume that the waiting time before executing an enabled call is sufficiently small, and its runtime sufficiently quick, that the implementation will always have completed any call it plans to complete before the end of the current phase.

Using the operational semantics developed in Section 3.2 we can compute the possible completed states of each phase, which are consistent with the actions of the implementation, denoted with  $Viable_i$  for phase  $i$ . Initially this is the set

$$Viable_0 \equiv \{\langle \epsilon, \emptyset, sa_0, ss_0, \epsilon \rangle\}$$

where  $sa_0$  is the initial state of the shared resource (defined by the initial rule),  $ss_0$  is the initial state of its scheduling policy, and  $\epsilon$  is the empty sequence.

We compute the set  $Viable_{i+1}$  from  $Viable_i$ , by computing the completed states  $completed(\cdot)$  for any state in  $Viable_i$  with any ordering of the calls in  $cmd_{i+1}$  as

input. Moreover, the set  $Viable_{i+1}$  is restricted to states such that the completed calls by the implementation  $finished_{i+1}$  coincide with the completed states predicted by the transition relation.

Formally then, we define  $Viable_{i+1}$  as

$$Viable_{i+1} \equiv \left\{ \begin{array}{l} s \in completed(\langle in_{i+1}, calls_i, sa_i, ss_i, \epsilon \rangle) \\ \text{such that} \\ \langle \epsilon, calls_i, sa_i, ss_i, out_i \rangle \in Viable_i \\ \wedge \quad in_{i+1} \in fac(cmd_{i+1}) \\ \wedge \quad to\_set(out(s)) = to\_set(finished_{i+1}) \end{array} \right\}$$

where  $fac(cmd_{i+1})$  is the set of sequences constructable from the set of calls in  $cmd_{i+1}$ , and  $to\_set(out_{i+1})$  is the set containing all the calls in the sequence  $out_{i+1}$ .

Note that even with the restriction of a single call per phase, the above set can have a size greater than one, as the scheduling specification may well permit multiple calls to execute at any point in time. Consider for instance a resource with two methods  $b$  and  $a$ , and in the two first phases one call to  $b$  is made in each, which block. Then in the third phase a call to  $a$  is made, which unblocks one of the calls to  $b$ , but the scheduling specification may well permit both to proceed.

However, note that in such a case, the *out* components of the resulting states are guaranteed to be distinct, as the order in which the calls complete will differ.

In our testing framework, however, the order in which completed calls occur in the *out* component cannot be observed, as explained in Section 4.1, and moreover, concurrent calls to the resource are permitted (leading to another race between applying operational semantics rules 2 and 3). This restriction on observational power has the effect that our testing framework must cope with an uncertainty regarding which state is the real state of the tested resource (and its scheduler).

We can thus summarize the testing procedure, for phase  $i+1$ , given the *viable* states at phase  $i$ , as follows.

1. Generate the calls  $cmd_{i+1}$  for phase  $i+1$  and execute them concurrently.
2. Record which calls have been completed by the implementation at the end of the phase as  $finished_{i+1}$ .
3. Calculate the next set of viable states,  $Viable_{i+1}$ , according to the above definition (from  $Viable_i$ ,  $cmd_{i+1}$ , and  $finished_{i+1}$ ).
4. If the size of  $Viable_{i+1}$  is 0, signal testing failure, since there is no execution of the test model which can explain the results of executing the implementation.

The computation of the set of viable states is done in a lazy manner, taking care not to generate all potential states at once, but rather computing  $completed(s)$  in a stepwise manner, discarding failed intermediate states at once, and merging intermediate identical states as soon as possible, to improve analysis efficiency. Nevertheless, in the worst-case scenario there may be an exponential number of viable states to explore. To

combat such situations we explore only tests of a limited size, where the number of concurrent calls are severely limited by design.

## QuickCheck

The basic functionality of the QuickCheck tool is rather simple: when supplied with a data term that encodes a Boolean property, which may contain universally quantified variables, QuickCheck generates a random instantiation of the variables, and checks that the resulting Boolean property is true. This procedure is by default repeated at most 100 times. If for some instantiation the property returns false, or a runtime exception occurs, an error has been found and testing terminates.

## Erlang

Quviq QuickCheck uses the Erlang functional programming language<sup>19,20</sup> to express correctness properties and test models. This does not mean that the tested software must be written in Erlang; a good interface library for C code has for example permitted the testing of AUTOSAR components and infrastructure on a commercial basis.<sup>21</sup> In this article we focus on testing control systems written in Java using the JavaErlang interface library (<https://github.com/fredlund/JavaErlang.git>).

## QuickCheck state machines

For checking ‘stateful’ code, QuickCheck provides a state machine library. Here the tested ‘object’ is not a simple Boolean property, but rather a sequence of function calls, each with an associated post condition that determines whether the execution of a call was successful or not. A QuickCheck state machine has a state, obviously, which can be understood as the model state of the system being tested. Given a model state, the library *generates* a suitable next API command, and proceeds to execute the call, checking after the call has completed whether the result was the expected one given the model state of the state machine. Next, a new model state is computed, and the generation of commands and their execution is repeated, until a test sequence of sufficient length has been generated and tested. In other words, the QuickCheck state machine acts as a *model* for the program being tested.

To use the state machine library a user has to supply a ‘callback’ Erlang module providing a set of functions with predefined names. The functions defined in the callback module are called by QuickCheck during test generation and test execution. The functions that should be implemented by a tester are enumerated below.

```
initial_state()
command(State)
```

```
precondition(State, Call)
next_state(State, Result, Call)
postcondition(State, Call, Result)
```

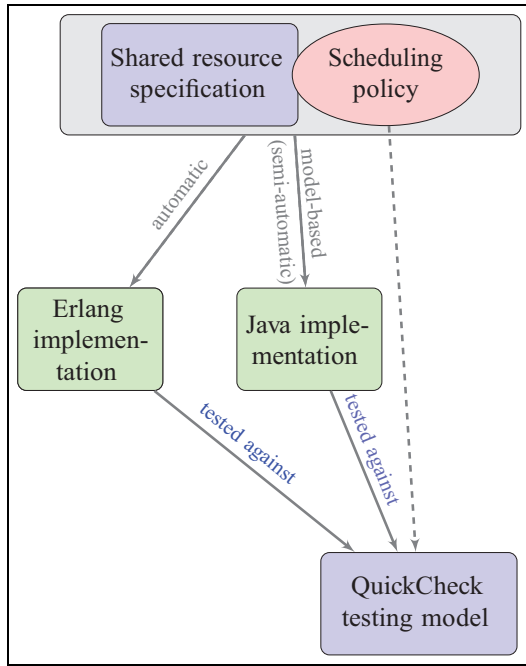
The model state is initialized by the `initial_state` function, and is updated by `next_state`. API calls are generated by the function `command`, which returns symbolic calls of the form `{call, Module Name, Function, Args}`, which are then executed. The `postcondition` function checks that the return value of a call is correct, considering the current model test state.

## Testing the warehouse resource

To test an implementation of the Warehouse resource using our prototype shared resource testing framework ([https://bitbucket.org/fredlund1/shared\\_resources\\_erlang](https://bitbucket.org/fredlund1/shared_resources_erlang)), there are four essential tasks to complete.

- Coding the warehouse shared resource (in Figure 1) using the syntax of the tool. For completeness the resulting description is included in Appendix A. As can be seen from the description, although expressed in a concrete programming language, the description is still rather concise and readable, strongly resembling the original specification.
- Deciding on which scheduling policy to use. Here we use a general scheduling policy, the ‘most liberal’ scheduling policy, which permits any enabled call to be executed at any time. For completeness the implementation of that scheduling in our framework is included in Appendix B.
- Implementing the call generator. That is, in each testing phase deciding how many concurrent calls to make, and deciding the format of each call. To do the generation a QuickCheck state machine is used. Below we explain in some detail how the generation of calls is implemented for the warehouse example.
- Implementing the ‘glue’ code which, for each generated call to the resource invokes the actual implementation being tested, and, moreover, observes which of the calls performed during the current testing phase has completed (waiting a specified time interval). This is a largely trivial task, which we will not concern ourselves with further in this article.

In Figure 6 we present a brief description of the whole approach. The framework relies on three components: (i) a shared resource implementation in Erlang, (ii) a scheduling policy expressing the set of enabled calls in a certain program point and (iii) a Java implementation that fulfils the presented policy. From that, QuickCheck creates a FSM based on the scheduling presented, and then it starts testing the given implementation by executing an *enabled* call in the Erlang implementation and replicating it in the Java implementation used. If an enabled operation can be performed in



**Figure 6.** Framework flowchart showing all components of the systems.

Erlang but it is not possible to execute it on the Java implementation, then an error has been found.

### Call generation

To produce more comprehensible tests we introduce the notion of a robot identifier, which is simply a natural number. In the model we extend the warehouse operations with a robot identifier as a first argument, in other words, a call is now `enterWarehouse(r, n, w)` where  $r$  is the robot identifier,  $n$  is the warehouse identifier, and  $w$  is the weight. However, before actually issuing the call to the implemented resource, the robot identifier is stripped. Thus robot identifiers are used only internally in the QuickCheck state model, and the resource specification need not change.

To generate meaningful commands for the warehouse example it is necessary to know the state of the robots. For example, if a robot is in warehouse 0 it makes no sense to generate a call for that robot asking to enter warehouse 5. The following then are the constraints which call generation should satisfy.

- A robot first enters warehouse 0, then exits warehouse 0 towards corridor 1, then enters warehouse 1, and so on. That is, there is a strict sequence of locations that each robot should attempt to pass through in sequence.
- If a call which concerns robot  $r$  is blocked, it makes no sense to issue another call concerning robot  $r$  until the first call is unblocked.
- Calls issued concurrently should concern distinct robots.

Unfortunately, there is in general no way to determine in advance which calls will block. For example, assuming two concurrent calls `enterWarehouse(1, 0, 900)` and `enterWarehouse(2, 0, 900)` corresponding to both robot 1 and robot 2 wanting to enter warehouse 0 with weight 900 (we assume a weight limit of 1000 for a warehouse) the implementation may choose any of these calls to complete (and the other will remain blocked). Thus, in the next testing phase, we cannot generate a command concerning either robot 1 (because it is blocked), or robot 2 (because it too may be blocked). For this reason we do not generate the complete test case, comprising all test phases, prior to testing. Rather the test case is built, *dynamically*, during *test execution*, using the knowledge obtained from actually issuing calls to the implementation regarding which calls blocked and which did not. In the situation above, for instance, we simply observe which call blocked and refrain from issuing new calls to that robot until it is unblocked, whereas we are free to issue new commands to the robot that succeeded in entering warehouse 0.

This call generation procedure is implemented using a QuickCheck state machine (see previous section for an introduction). The state of the machine keeps track of blocked calls (to prevent issuing of a call to a robot with a blocked call), and which robots are present in which corridors and warehouses. In the following we abbreviate `enterWarehouse` as `enter` and `exitWarehouse` as `exit`.

The actual commands to generate in a test phase are chosen as a random, small set of concurrent calls, where each call is chosen randomly from all possible commands. As an example, we show below a QuickCheck generator that is capable of generating random commands to exit a warehouse (towards the corridor), that is, `exit(r, n, w)`, using the current model state:

```

oneof
([call, warehouse, exit, [R, N, W] ||
  N <- warehouses(),
  {R, W} <- warehouse(N, State),
  not (lists:member
        (R, blocked(State)))
]).

```

(A QuickCheck generator is a function that is capable of, according to some probability distribution, generating an infinite number of elements for some Java datatype. The generator `int()`, for example, can generate random integers, and `list(int())` generates lists of random lengths, containing random integers.) The above code comprises a *generator* in QuickCheck terminology, which is a piece of code which can be called repeatedly to generate random calls to the `exit` method. Note that which commands the generator can generate depends crucially on the state parameter (`State`). In the above fragment, `warehouses()` is a list of warehouse identifiers  $(0, 1, \dots)$ , and  $N$  is bound to

a random such warehouse. Then  $\{R, W\}$  corresponds to one (if any) of the robots in warehouse  $N$  (according to the state), and its associated weight. Moreover, it is a requirement that the robot chosen ( $R$ ) cannot be mentioned in a blocked call (last line). Thus, the above code fragment randomly selects (using the QuickCheck generator `oneof`) a robot located in any warehouse, which is not mentioned in a blocked call, and issues an exit command for that robot.

The full command generator also generates enter commands; we cut down on the number of possible commands by enforcing that robots enter warehouse 0 with sequentially increasing robot identifiers, starting with 0, up to some small maximum (10). To increase the possibility that the sum of weights in a warehouse is exactly the maximum weight in a warehouse (normally 1000), starting weights for robots are chosen randomly using the QuickCheck generator `?LET(X, eqc_gen:choose(1,11),X*100)`, in other words, the generator first chooses a random integer between 1 and 11, and then multiplies it by 100. Thus possible weights are 100, 200, ..., 1100.

As an example, the following set of (concurrent) calls could be generated from the initial model state: `enter(0,0,300)`, `enter(1,0,700)` and `enter(2,0,300)`. Note that the concurrent calls concern different robots to prevent interference.

### Experimental validation of the testing framework

To validate the approach we used the testing framework described in this article to test 103 Java-based implementations of the warehouse shared resource. The artefacts necessary for testing are the shared resource specification of the warehouse in Appendix A, the specification of the scheduling policy in Appendix B, and finally the QuickCheck state machine which computes the test cases developed according to the scheme described in Section 6.1.

The implementations that were tested using the framework were written by undergraduate students attending a course on concurrency at the Technical University of Madrid. The students were given the formal specification of the shared resource, and an informal description of the required scheduling policy. To implement the resource they were required to use a particular concurrency construct,<sup>22</sup> which is an improvement of Java's native *locks and conditions* library in that it is not needed to test the concurrency precondition using a while loop.

Before we ran the QuickCheck-based test on the student programs, the students had already tested their solutions using a manually crafted JUnit test suite, which was supposed to detect implementations which violated the safety and progress criteria relevant for the warehouse specification. All 103 implementations that were tested using our testing framework had already successfully passed the JUnit test suite. Moreover, the students had a strong incentive to hand in good

solutions, as the warehouse implementations were graded, and these grades were factored into the final course grade.

Although the task may not appear overly difficult, the results of our testing using QuickCheck were, at least to us, surprising. Of the 103 solutions tested, we found errors in 55 of them, in other words, 54.4% of the solutions handed in contained at least one error, indicating that the particular JUnit test suite for this example was not particularly good at finding errors, and that the QuickCheck-based testing approach was much more successful. (Although we had not evaluated the quality of the JUnit test suite using any coverage measure, it was believed before this experiment was conducted that the suite was quite good at finding implementation errors.)

We separated the testing of the implementations into three test rounds, where during a test round individual tests were executed which attempted to identify failures roughly corresponding to one of the failure categories enumerated in Section 4.

The first test round (*round1*) tried to identify implementations which failed a basic safety criterion, for example an implementation which admits a robot into a warehouse even when the total resulting weight exceeds the limit, or an implementation which admits a robot into a corridor even when the corridor is already occupied by another robot (corresponding to errors of type **bad-spec-impl** in Section 4). The second test round (*round2*) additionally tested whether there are calls (to enter or exit a warehouse) which the model permits, but which the implementation blocks. That is, the implementation does not satisfy a liveness condition (corresponding to errors of type **bad-sched**). Finally, during test *round3*, in contrast to *round1* and *round2*, tests were executed which consisted of multiple concurrent calls to the resource, in order to detect possible incorrect uses of the basic Java concurrency mechanisms.

The results of the testing using our framework are summarized in Table 1. Note that test round *round3* was run only if no errors were detected during test *round1* or test *round2*. To properly interpret the results, note that the properties tested during the three test rounds are not mutually exclusive. That is, the test cases tested during *round2* are stricter than those in *round1* (i.e. testing both safety and liveness), whereas *round3* is

**Table 1:** Test results for the students' implementations of the warehouse example, classified by test rounds.

PASSED	FAILED			
*48	55			
	<i>round1</i>	<i>round2</i>	<i>round3</i>	number
	✓	–		2
	–	✓		13
	✓	✓		38
	–	–	✓	2

```

1.  *** Error: there are calls that have
2.  been completed by the implementation
3.  which cannot be completed by the
4.  model
5.
6.  Final test state:
7.    w(0)=4:700; corr(1)=2:700
8.    w(1)=0:300,1:300; corr(2)={}
9.    w(2)=;corr(3)={} w(3)={};
10.
11. Final model state:
12.   w(0)=700; corr(1)=true
13.   w(1)=600; corr(2)=false
14.   w(2)=0; corr(3)=false
15.   w(3)=0;
16.
17. Schedule state:
18.   Void
19.
20. postcondition false for call(s)
21.   exit(0,1,300)
22. Test failed with reason
23.   {postcondition,false}
24.
25. Command sequence:
26. -----
27. start
28. << enter(0,0,100) >> -- unblocks 0
29. << exit(0,0,100),
30.   enter(1,0,200) >> -- unblocks 0, 1
31. << enter(0,1,300),
32.   exit(1,0,200) >> -- unblocks 0, 1
33. << enter(2,0,700) >> -- unblocks 2
34. << exit(2,0,700) >>
35. << enter(1,1,300),
36.   enter(4,0,700) >>
37.       --unblocks 1, 2, 4
38. << enter(2,1,900) >>
39. << exit(0,1,300) >> -- unblocks 0, 2

```

#### Sample error report.

more strict than both *round2* and *round1*, as its test cases test both safety and liveness under the added complication of (possibly) concurrent calls. As the testing is randomized, there is a chance that although an implementation error is, say, first detected during test *round2*, it may be still be a safety error which due to the nature of random test generation, was, through chance, not detected during *round1*. In the table ‘PASSED’ are those implementations (48) in which testing failed to detect an error, and those under the heading ‘FAILED’ (55) had at least one bug. The Figure moreover indicates which round spotted an error. Thus, as an example, among the failed implementations, row 2 shows that there were 13 implementations that failed test *round2* (and did not fail *round1*), while the last row tells us that two additional implementations were found

erroneous using *round3* whose errors were not detected by either *round1* or *round2*.

Figure 7 shows an example of an automatically generated error-detection report (note that the detection report uses customized display routines for test states and model states, and for the printing of unblocked calls).

The command sequence shows the sequence of calls that provoked the testing error. Concurrent calls in a round are enclosed within ‘<< >>’. After each round, the identities of the robots in unblocked calls are printed. That is, the first round consists of a single command `enter(0,0,100)` corresponding to robot 0 asking for permission to enter warehouse 0 with weight 100. Moreover, the call succeeded, as shown by the ‘-- unblocks 0’ indication.

The indicated error is that at least one of the unblocked calls (0 and 2) by the last call could not be unblocked by the model, but was incorrectly unblocked by the implementation. The final test state and the final model state, shown in the piece of code above (lines 6 and 11), represent the state of the QuickCheck state machine generating the calls to execute, and the shared resource state, respectively, before the last executed call (which caused the implementation bug to be detected).

In the call sequence, and using the state information, we can read that robot 0 was given permission to exit warehouse 1 with a weight of 300, and moreover that an earlier call `enter(2,1,900)` was unblocked too. Intuitively, since the total weight in warehouse 1 was 600 before the last call, when robot 0 left the warehouse the total weight was reduced to 300. However, since robot 2 also entered the warehouse with a total weight of 900, there was no way for the model to mimic the behaviour of the implementation as the total weight in warehouse 1 after the unblocking of the calls would be  $600 - 300 + 900 = 1200$ , which exceeds the permitted maximum weight 1000. In other words, testing has detected a discrepancy between the actions of the model and the implementation, that is, a safety-critical bug was found in the implementation as it is supposed to faithfully implement the model.

Manually crafted tests were used to effectively find errors related to understanding the specification incorrectly, as well as blocking processes according to concurrency preconditions. Nevertheless, programming that part of the system is relatively straightforward. It is much more difficult to correctly program the process unblocking functionality. QuickCheck-based tests are very effective in detecting incorrect process signalling. Since the scenarios that lead to an error were shrunk by QuickCheck, the students could mentally execute them and find the most common mistakes: processes with different concurrency preconditions were blocked in the *same queue*; and two processes were signalled because their concurrency preconditions were true but one invalidated the precondition of the other.

## Conclusions and future work

Concurrent programming is a challenging task and nondeterministic execution makes testing harder than in the case of sequential software. Fortunately, the use of formal specifications and model-driven methodologies can help programmers in avoiding some of its pitfalls, by helping validate concurrency requirements, bringing structure and discipline in the use of language constructs to the implementation of the more delicate parts of a concurrent system and, as we show in this work, generating meaningful test cases.

We have presented a framework for testing concurrent safety-critical systems which have been designed and implemented from shared resources, a high-level notation for specifying process interaction. A model

implementation of the shared resource is defined using the Erlang language and run in parallel with the actual implementation being tested. The QuickCheck environment for property-based testing is responsible for generating test sequences and detecting whether the behaviour of the real code deviates from that of the model one, thanks to its native support for state machines. Currently, the framework is prepared for testing Java implementations, but, as the approach is black-box, there is no fundamental limitation that prevents the framework from dealing with other programming languages. The framework is available at [https://bitbucket.org/fredlund1/shared\\_resources\\_erlang](https://bitbucket.org/fredlund1/shared_resources_erlang).

Other contributions of this work include a parametric operational semantics for shared resources with scheduling annotations, which is the foundation of the execution of the model Erlang execution, and a rough but helpful classification of the most typical error situations and how to guide the test generation process in order to maximize the accuracy of test suites.

We have tested our framework on a prototypical safety-critical system used to teach concurrency at our university. This allowed us to try it against around 100 different versions of the same system. The experimental results show that our automated semi-random test generation scheme is able to reveal errors that passed undetected by a carefully crafted set of JUnit tests.

Items for future work include providing the functionality of deriving individual test cases (and indeed entire test suites). This can already be achieved using the approach explained here, except that the execution of a generated test case need not be deterministic, but instead depend on the particular implementation. Thus such a 'pre-generated' test case may have to be aborted in mid-run because an invoked operation may be nonsensical (e.g. if a robot desires to exit a warehouse before it has been given permission to do so). In contrast, using the approach adopted in this article we do not have to abort test cases in mid-run, as the test-case generation is *steered* by the actual implementation being tested.

In the present paper we have focused on systems where several threads interact on a *single* shared resource. While this is not a limitation from a theoretical perspective, that is, from the point of view of expressing a discrete transition system, it is true that the code generated for single-resource designs can be less efficient than that generated for decentralized, many-resource designs. We have gained some experience with decentralized designs and we have shown how to implement 'low contention' concurrent algorithms with them. However, we have not yet devised a general method for verifying or testing many-resource systems. In these systems, besides having the logical specification for each resource, special care has to be taken to preserve certain global or system-wide invariants. Improving our techniques for reasoning about and testing many-resource systems is on our short-term agenda.



## Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

## Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was partially funded by the European Commission (FP7 project ICT-2011-317820 (PROWESS)), by the Comunidad de Madrid (grant S2013/ICE-2731 (N-Greens Software)), by the Spanish MINECO (project TIN2012-39391-C04-03 (Strong Soft)), and by the Spanish Ministry of Industry, Energy and Tourism and the ARTEMIS JU under grant no.~295373 (nSafecer).

## References

- Herranz A, Mariño J, Carro M, et al. Modeling concurrent systems with shared resources. In: Alpuente M, Cook B and Joubert C (eds) *Formal methods for industrial critical systems (Lecture Notes in Computer Science, vol. 5825)*. Berlin: Springer-Verlag, 2009, pp. 102–116.
- Arts T, Hughes J, Johansson J, et al. Testing telecoms software with Quviq QuickCheck. In: *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, Portland, OR, 2006, pp. 2–10.
- Carro M, Mariño J, Herranz A, et al. Teaching how to derive correct concurrent programs from state-based specifications and code patterns. In: Dean CN and Boute R (eds) *Teaching formal methods (Lecture Notes in Computer Science, vol. 3294)*. Berlin: Springer, 2004, pp. 85–106.
- Mariño J and Alborodo RNN. A model-driven methodology for generating and verifying CSP-based Java code. In: Welch PH (ed.) *Communicating process architectures 2015*. Oxford, UK: Open Channel Publishing Ltd., 2015.
- Claessen K and Hughes J. QuickCheck: A lightweight tool for random testing of Haskell programs. In: *Proceedings of the fifth ACM SIGPLAN international conference on functional programming*, New York, NY, 2000, pp. 268–279.
- Staats M, Gay G, Whalen M, et al. On the danger of coverage directed test case generation. In: De Lara J and Zisman A (eds) *Fundamental approaches to software engineering (Lecture Notes in Computer Science, vol. 7212)*. Berlin: Springer, 2012, pp. 409–424.
- Binder RV, Legeard B and Kramer A. Model-based testing: Where does it stand? *Commun ACM* 2015; 58(2): 52–56.
- Ponce de León H, Haar S and Longuet D. Model-based testing for concurrent systems with labelled event structures. *Softw Test Verif Reliab* 2014; 24(7): 558–590.
- Carver R and Lei Y. A modular approach to model-based testing of concurrent programs. In: Lourenço J and Farchi E (eds) *Multicore software engineering, performance, and tools (Lecture Notes in Computer Science, vol. 8063)*. Berlin: Springer, 2013, pp. 85–96.
- Ulrich A and König H. Specification-based testing of concurrent systems. In: Mizuno T, Shiratori N, Higashino T, et al. (eds) *Formal description techniques and protocol specification, testing and verification*. New York, NY: Springer, 1997, pp. 7–22.
- De Nicola R and Hennessy MCB. Testing equivalences for processes. *Theor Comput Sci* 1984; 34(1): 83–133.
- Brinksma E and Tretmans J. Testing transition systems: An annotated bibliography. In: Cassez F, Jard C, Rozoy B, et al. (eds) *Modeling and verification of parallel processes (Lecture Notes in Computer Science, vol. 2067)*. Berlin: Springer, 2001, pp. 187–195.
- Betin-Can A, Bultan T, Lindvall M, et al. Application of design for verification with concurrency controllers to air traffic control software. In: *Proceedings of the 20th IEEE/ACM international conference on automated software engineering*, New York, NY, 2005, pp. 14–23.
- Yavuz-Kahveci T and Bultan T. Specification, verification, and synthesis of concurrency control components. In: *Proceedings of the 2002 ACM SIGSOFT international symposium on software testing and analysis*, New York, NY, 2002, pp. 169–179.
- Larsen KG, Mikucionis M and Nielsen B. Online testing of real-time systems using UPPAAL. In: Grabowski J and Nielsen B (eds) *Formal approaches to testing of software (Lecture Notes in Computer Science, vol. 3395)*. Berlin: Springer.
- Cheung KS and Chow KO. Process-based design verification for systems involving shared resources. In: *IEEE Asia-Pacific conference on services computing*, 2006, pp. 99–106.
- Herlihy MP and Wing JM. Linearizability: A correctness condition for concurrent objects. *ACM Trans Program Lang Syst* 1990; 12(3): 463–492.
- Courtois PJ, Heymans F and Parnas DL. Concurrent control with readers and writers. *Commun ACM* 1971; 14(10): 667–668.
- Armstrong J, Virding R, Wikström C, et al. *Concurrent programming in Erlang*. Englewood Cliffs, NJ: Prentice-Hall, 1996.
- Cesarini F and Thompson S. *Erlang programming – A concurrent approach to software development*. Sebastopol, CA: O'Reilly Media, 2009.
- Svenningsson R, Johansson R, Arts T, et al. Testing AUTOSAR basic software models with QuickCheck. In: Pavese F, Bár M, Filtz J-R, et al. (eds) *Advanced mathematical and computational tools in metrology and testing IX*. Singapore: World Scientific, 2012, pp. 391–395.
- Herranz A and Mariño J. A verified implementation of priority monitors in Java. In: Beckert B, Damiani F and Gurov D (eds) *Proceedings of the 2nd international conference on formal verification of object-oriented software (FoVeOOS'11), revised lectures (Lecture Notes in Computer Science, vol. 7421)*. New York, NY: Springer, 2012, pp. 160–177.

## Appendix

### A. The warehouse shared resource coded in Erlang

```
-module(robots) .
-export([init/1,pre/2,cpre/2,post/2]) .
-record(robots,
{
    num_naves, %% constant !
    max_weight, %% constant !
```

```

    corridors,
    warehouses
  })
init([NumNaves,MaxWeight]) ->
#robots
{
  num_naves=NumNaves,
  max_weight=MaxWeight,
  warehouses=
    lists:map
    (fun (I) -> {I,0} end,
    lists:seq(0,NumNaves-1)),
  corridors=
    lists:map
    (fun (I) -> {I,false} end,
    lists:seq(1,NumNaves-1))
}.
pre(enter,[_R,N,W],State) ->
is_integer(N) andalso (N>=0)
andalso (N<State#robots.num_naves)
andalso is_integer(W)
andalso (W>=0);
pre(exit,[_R,N,W],State) ->
is_integer(N) andalso (N>=0)
andalso (N<State#robots.num_naves)
andalso is_integer(W)
andalso (W>=0).
cpre(enter,[_R,N,W],State) ->
(weight(N,State) + W) = <
State#robots.max_weight;
cpre(exit,[_R,N,W],State) ->
(N==(State#robots.num_naves-1))
orelse (not(occupied(N+1,State))).
post(enter,[_R,N,W],State) ->
if
  N==0 ->
    add_weight(W,N,State);
  true ->
    add_weight
    (W,N,remove_robot(N,State))
end;
post(exit,[_R,N,W],State) ->
if
  N==State#robots.num_naves-1 ->
    add_weight(-W,N,State);
  true ->
    add_weight
    (-W,N,add_robot(N+1,State))

```

```

end.
add_weight(W,N,State) ->
{_,OldWeight =
  lists:keyfind
  (N,1,
  State#robots.warehouses),
  State#robots
  {warehouses=
    lists:keystore
    (N,1,State#robots.warehouses,
    {N,OldWeight+W})}.
remove_robot(N,State) ->
State#robots
{corridors=
  lists:keystore
  (N,1,State#robots.corridors,
  {N,false})}.
add_robot(N,State) ->
State#robots
{corridors=
  lists:keystore
  (N,1,State#robots.corridors,
  {N,true})}.
weight(N,State) ->
{_,Weight =
  lists:keyfind
  (N,1,State#robots.warehouses),
  Weight.
occupied(N,State) ->
{_,Occupied} =
  lists:keyfind
  (N,1,State#robots.corridors),
  Occupied.

```

### B. The permissive scheduling policy

```

-module(always).
-export([init/2,waiting/3,
enabled/4,post_waiting/4]).
init(_,_) ->
  void.
waiting(_Call,State,_) ->
  void,State.
enabled(_Call,_CInfo,_State,_) ->
  true.
post_waiting(_Call,_CInfo,State,_) ->
  State.

```