# A Model-Driven Approach to Teaching Concurrency

MANUEL CARRO, Universidad Politécnica de Madrid and IMDEA Software Institute
ÁNGEL HERRANZ and JULIO MARIÑO, Universidad Politécnica de Madrid

We present an undergraduate course on concurrent programming where formal models are used in different stages of the learning process. The main practical difference with other approaches lies in the fact that the ability to develop correct concurrent software relies on a systematic transformation of formal models of inter-process interaction (so called *shared resources*), rather than on the specific constructs of some programming language. Using a resource-centric rather than a language-centric approach has some benefits for both teachers and students. Besides the obvious advantage of being independent of the programming language, the models help in the early validation of concurrent software design, provide students and teachers with a *lingua franca* that greatly simplifies communication at the classroom and during supervision, and help in the automatic generation of tests for the practical assignments. This method has been in use, with slight variations, for some 15 years, surviving changes in the programming language and course length. In this article, we describe the components and structure of the current incarnation of the course—which uses Java as target language—and some tools used to support our method. We provide a detailed description of the different outcomes that the model-driven approach delivers (validation of the initial design, automatic generation of tests, and mechanical generation of code) from a teaching perspective. A critical discussion on the perceived advantages and risks of our approach follows, including some proposals on how these risks can be minimized. We include a statistical analysis to show that our method has a positive impact in the student ability to understand concurrency and to generate correct code.

## 1. INTRODUCTION

Teaching concurrent programming at the undergraduate level is a challenging task. Reasonably enough, early courses offer a simplified, nonreactive, input-output view of computation. Students often face interaction and temporal issues for the first time in a concurrency course, and many of them find it difficult to visualize concurrent

The authors were partially supported by grant S2009TIC-1465 *PROMETIDOS-CM* from the Madrid regional government. J. Mari˜no and ´A. Herranz were additionally supported by Spanish MINECO grant TIN2009- 14599-C03-03 *DESAFIOS10*. M. Carro was also supported by Spanish MINECO grant TIN-2008-05624 *DOVES*.

execution. This can lead to early frustration and negative attitudes toward the subject. To add to this difficulty, concurrency is seldom taught in the lower-level undergraduate curriculum [Feldman and Bachus 1997], which makes some departments reluctant toward it. However, it is now clear (as advanced by Yeager [1991]) that concurrent programming is of utmost importance.

Well-known issues regarding the development of concurrent software also appear when teaching the subject, for instance, the intrinsic difficulty of testing and debugging concurrent code, which makes black-box approaches of limited applicability. While it is hard for instructors to assess the correctness of even simple programs (and to grade them!), it is even harder for students to do so. Also, there is unequal support of concurrency in programming languages. The first language chosen for the introductory courses might not be suitable for concurrency and introducing a new one for just one subject is usually out of the question.

These observations led us to develop a teaching method based on formal models, which has been in use essentially untouched for 15 years. Its distinguishing feature is that the code relevant for process communication and synchronization is obtained by systematic transformation of a formal model of interprocess interactions that is programming language-independent. In 1997, Ada95 [Taft et al. 2001] was the common programming language used in most of our courses and the method used two idiom-based transformations, one for *protected objects* and another one for *rendez-vous*. Java is the current choice and three idioms are used: one for *synchronized methods*, another one using our own implementation of *locks and conditions*, and a third one using the JCSP [Welch et al. 2007][1] library. Using language-independent formal abstractions allows us to factor out some issues and spend less time explaining language specific details.

Beyond language independence, model-based development helps in the early validation of concurrent software design, provides students and teachers with a *lingua franca* that greatly simplifies communication at the classroom and during supervision, and helps in automatically correcting assignments. Of course, there are some risks in a naïve application of the model-driven approach, and a number of assessment activities have been developed to minimize them.

We are not aware of any other undergraduate course on concurrent programming following a similar approach. Unfortunately, a gap between formal methods and the early courses in Computer Science still persists. However, we hope that some factors, like the availability of model-checking tools and their impact on the teaching of concurrency, regardless of the teaching method (see, for instance, Ben-Ari [2009]), will help in correcting this situation. Also, there is a renewed interest in discussing the challenges of teaching concurrency, exemplified by recent workshops such as the Workshop on Teaching Concurrency[2] and the Workshop on Curricula for Concurrency and Parallelism [Steele and Saraswat 2009; Saraswat and Bruce 2010]. Many of the contributions in these (and other) workshops emphasize points with which our experience agrees: the need for tool support [Ben-Ari 2004; Sadowski et al. 2011], the natural connection between concurrency and data abstraction courses [Grossman and Anderson 2012], the need for safe idioms to avoid error-prone features of programming languages [Carro et al. 2004; Grossman and Anderson 2012], and others. Also, a number of concurrency-related courses have begun to introduce more or less explicitly the model-based tag (e.g., the undergraduate concurrency course at Stony Brook[3] and the course described in Brabrand [2008]), following a trend in other software construction disciplines.

[1]JCSP is a Java library that provides a concurrency model integrating Hoare's CSP and Milner's $\pi$-calculus. See http://www.cs.kent.ac.uk/projects/ofa/jcsp/.

[2]TeaConc2006: http://www.uninova.pt/gres/teaconc2006/. TeaConc2007: http://www.uninova.pt/teaconc2007/.

[3]http://www.cs.sunysb.edu/undergrad/cse_courses/cse375.html.

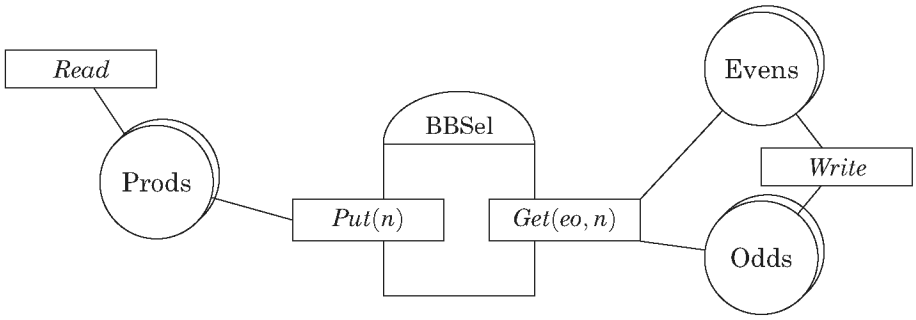


Fig. 1. Bounded buffer with selection: components.

Section 2 provides a high-level description of the model-driven methodology we apply, emphasizing the differences between an ideal method that would be applied for real software development and the actual method for classroom usage. Basic information on the course structure is given. Section 3 summarizes the different outcomes obtained from the use of formally specified resources, and Section 4 dissects those outcomes to analyze the pros and cons of the model-driven approach from a teaching perspective. Section 5 is a statistical analysis to show that our method has a positive impact in the students' ability to understand concurrency and to generate correct code. Section 6 concludes the article.

## 2. TEACHING A RESOURCE-CENTRIC MODEL-DRIVEN DEVELOPMENT

Ideally, enough time should be devoted to teaching all the stages of the software development cycle, which in our case is centered around the notion of a concurrent shared resource (Section 2.2). It is, of course, not necessary to teach the workflow activities in the order in which they happen: It may be pedagogically more fruitful to start with problems that appear at the end of the development cycle and show how taking corrective actions at its beginning (such as thorough analysis and careful design) can reduce trouble later on.

Besides this general consideration, constraints regarding time and student background force us to adapt our teaching strategy, skip over some parts, and transform others (namely, the design phase) into a closely related activity.

### 2.1. Architectural Components

The approach we propose is part of a development methodology that was first presented in Carro et al. [2004] and that we summarize here. In our proposal, a concurrent system is made up of two different classes of components.

—A set of active components (processes) that react to the external world (e.g., reading sensors and acting on physical elements) and interact through calls to operations of a concurrent shared resource.
—A concurrent shared resource, which can be seen as a concurrent abstract data type whose operations implement changes to some object state (which can, therefore, be used to communicate processes) and take care of synchronization by means of suspending the caller process. In some sense, they resemble some of the characteristics of the classical capsules for C++ [Gehani 1993]. Resources act as the unique communication and synchronization point between processes.

These two types of components (portrayed in Figure 1) mark a clear separation of concerns: Processes (Prods, Evens, and Odds) interact with the world (*Read*,

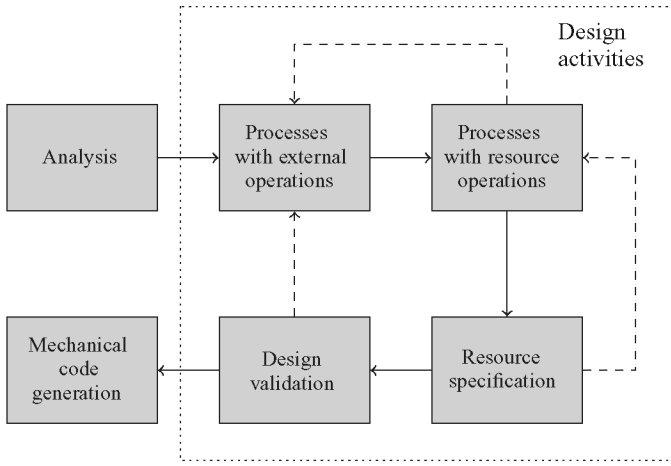


Fig. 2. Sketched workflow for a development approach based on shared resources.

*Write*), but they do not synchronize directly with each other; synchronization happens through suspension/resumption of calls to the methods of the shared resource (BBSel). Processes, therefore, do not suspend/resume voluntarily; in fact, processes have no notion of being or not being suspended; therefore, we can reason with a single process as if it were a purely sequential computation. On the other hand, a shared resource implementation will check whether conditions for method calls to proceed hold, but they have no direct knowledge of what the processes exactly do.

Processes can have private data and acquire data from the outside, but any communication among processes happens necessarily through the shared resource. Processes commonly take some decisions on external data and the outcome of the interaction with the shared resource, therefore featuring some (limited) algorithmic behavior, while shared resources often perform limited bookkeeping.

### 2.2. A Development Workflow with Shared Resources

Figure 2 sketches the stages of the workflow we use. After an initial analysis phase, which aims at identifying necessary concurrency, necessary sequentiality, and coarse-grained interactions between components, processes are identified together with the data that they need to communicate. External operations and their call protocols act as a basis on top of which an initial skeleton of the processes can be built with the help of some common sense and simple rules (e.g., if two external operations have to be called always in sequence, it makes sense to call them both from the same process). Process skeletons are then enriched with operations on the shared data. These operations are necessary to coordinate the processes and to transfer data among them through the shared resource. Finally, the shared resource itself is specified by:

—stating its operations and their types;
—identifying the local variables (i.e., the state) of the resource;
—defining the synchronization conditions for the operations in the resource (i.e., when the operations can proceed). Unlike what happens in some programming languages, these conditions (also called operation guards) can depend on both the internal resource state and the values of the parameters of the operations;
—defining the effect of successfully completed operations on the state of the resource and on the output parameters of the operations.

The code is derived from the processes and the shared resources as follows.

—Translating the processes into the target language. This is usually an easy task, since they are designed as sequential computations with no resorting to peculiarities of particular programming languages.
—Translating the shared resource to the target language. This needs capabilities to express conditional synchronization between processes and to ensure the absence of race conditions in the access to the data encapsulated by the resource. The translation of the synchronization primitives can be made mechanically, as their semantics is kept simple by design. This translation was presented in detail in Carro et al. [2004] with Ada as the target language. Note that we are not concerned here with the translation of data structures and sequential components: In our experience, and because they are also described using a specification language, coding patterns can be applied in many cases.

During this process, it may be necessary to revisit previous stages (marked with dashed arrows) if inconsistencies are found at any moment of the development, but especially when fully validating the design.

### 2.3. Shared Resources and Shared Memory

We want to note that we are not committing to any particular implementation mechanism for the shared resource. Although the word *shared* may suggest using constructs devised for shared memory architectures (e.g., monitors [Hoare 1974] or similar), message passing can be used by implementing the shared resource as a separate process that communicates with the rest of the processes in the system [Carro et al. 2009].

In this case, the semantics of the shared resource and the simplicity of the process language make it possible to write wrappers around the code of the resource operations to hide the use of messages/channels. The resource would then receive messages corresponding to calls to operations, act accordingly on private variables implementing the shared resource state, and send messages back with the result of finished operations—but in a fashion completely transparent to the client processes.

### 2.4. Resource-Centric Design

In the design stage, shared resources must be understood as passive[4] components that encapsulate data with well-defined interface and semantics. The most relevant properties of the shared resources are as follows.

—*Serializability.* For any set of simultaneous invocations of several of its operations, there is an equivalent sequential execution of the same operations, which, starting from the same initial state, leads to the same final state. In other words, any correct implementation of a shared resource must guarantee the absence of race conditions.
—*Conditional synchronization.* Besides mutual exclusion in the access to the resource, a process invoking an operation suspends when its guard evaluates to false. Once the guard holds, the call to the operation is allowed to proceed. We do not establish when it actually proceeds if several enabled operations compete.
—*Synchronous invocation.* A process invoking an operation is blocked until the post-condition established by the operation is made true.

[4]This has strong implications on the architecture: Two resources cannot communicate directly; they need to be connected through an intermediate process.

```
// Producer loop:
while (true)
{
  data = Read();
  b.put(data);
}


// Even consumer loop:
while (true)
{
  data = b.get(even);
  Write(data);
}


// Odd consumer loop:
while (true)
{
  data = b.get(odd);
  Write(data);
}
```

**CADT** BBSel
**OPERATIONS**
**ACTION** Put: $\mathbb{N}[i]$
**ACTION** Get: $\mathbb{N}[i] \times \mathbb{N}[o]$

**SEMANTICS**
**DOMAIN:**
**TYPE:** $BBSel = \text{seq}\,\mathbb{N}$
**INVARIANT:** $\#\text{self} \leq MAX$

**INITIAL:** $\text{self} = \langle\rangle$

**CPRE:** $\#\text{self} < MAX$
$Put(n)$
**POST:** $\text{self}^{\text{out}} = \text{self}^{\text{in}} \frown \langle n \rangle$

**PRE:** $eo \in \{0, 1\}$
**CPRE:** $\#\text{self} > 0 \wedge (\text{self}(1) \bmod 2 = eo)$
$Get(eo, n)$
**POST:** $\text{self}^{\text{out}} = \text{tail}(\text{self}^{\text{in}}) \wedge n^{\text{out}} = \text{self}^{\text{in}}(1)$

Fig. 3. Processes and bounded buffer with selection shared resource.

### 2.5. Example: A Bounded Buffer with Selection

With the help of the example in Figure 1, we will see how encapsulated data and the behavior of operations are specified using shared resources. We consider two types of processes: producers (Prods), which insert natural numbers read from the external world into the buffer, and consumers, which remove elements from it. In turn, there are two types of consumers: those that only want odd numbers (Odds) and those that only want even numbers (Evens). Producers suspend if the buffer is full and can proceed otherwise. Consumers suspend if the buffer is empty or if the first element in the buffer does not match the requested parity. The suspension conditions are not explicit in the processes: They are instead encoded in the specification of the shared resource (BBSel).

The formal specification of shared resources (Figure 3, right, in our example) is structured in three parts: the interface declaration (OPERATIONS), the internal state definition (DOMAIN), and the specification of the behavior of operations (PRE-POST). The language uses first-order logic formulas with a Z-like mathematical toolkit [Spivey 1992]. To be more precise, the language we use can, in practice, be seen as a specification that admits easily a first-order logic semantics (which makes reasoning with first-order logic tools possible) but where the style of specifications stays, on purpose, close to that of a single-assignment procedural language. The Z mathematical toolkit can be seen as calls to libraries implementing a convenient set of data structures, which makes it easy to translate the specification into a procedural/OO programming language. On the other hand, and as mentioned elsewhere, the shape of the specification makes it amenable to be translated into TLA+ with little to moderate effort (Section 3.1).

The bounded buffer specification of BBSel declares an interface with two operations (ACTION): *Put* and *Get*. *Put* has an input ($[i]$) formal parameter of type $\mathbb{N}$ (naturals). *Get* has two formal parameters: an input parameter ($\mathbb{N}[i]$) to indicate which kind of data the consumer wants (0 for even, 1 for odd) and an output parameter ($\mathbb{N}[o]$) where the data is returned to the consumer.

The description of the internal state is given by the definition of the domain of BBSel (TYPE). The domain defines the main type (a sequence of naturals in this case, N($\mathbb{N}$)) and an invariant, which must be true before any operation is executed and is required to hold after it finishes. In our example, the only invariant is the boundedness of the buffer, which is modeled by stating that the sequence length (#self) has to stay below a limit. The initial state for the resource (an empty sequence in this case) is specified with the formula after the clause INITIAL.

Three first-order logic formulas are used to specify the behavior of the operations: a precondition (PRE), a guard or concurrency precondition (CPRE), and a postcondition (POST).

The concurrency precondition (CPRE) of an operation is a logic formula that, using the state of the shared resource (represented by the variable self) and the actual operation arguments, states if the operation can proceed. The postcondition (POST) of an operation is a before-after predicate that relates the state of the resource and the actual arguments before (decorated with superscript "*in*") and after (decorated with superscript "*out*") the operation is executed. Comments in natural language can also be added to clarify the intended semantics of the operations. Optionally, a sequential precondition (PRE) with the standard meaning can be added for each operation.

In our example, an invocation to *Put* can proceed if the buffer is not full (#self $\leq$ MAX), and it modifies the resource state to store the received item at the end of the sequence ($r^{\text{out}} = r^{\text{in}} \frown \langle n \rangle$, where "$\_ \frown \_$" stands for sequence concatenation). An invocation to *Get* can proceed if the buffer is nonempty (#self $> 0$) and the element candidate to be removed has the same parity required by the process ($eo = even \implies r(1) \mod 2 = 0$). On success, the buffer is modified to remove its first element (recall that items were inserted at the end of the sequence).

### 2.6. Teaching: The Real World

Two issues are key in shaping the contents and approach of the course: the background of the students and the availability of time. On the one hand, it is necessary that students feel comfortable with coding simple sequential algorithms, are able to understand how data abstractions work without having an implementation to consult, and are mature enough to understand the problems associated with concurrent execution. The course is currently taught in the fourth semester (in the second part of sophomore year) of a CS degree.[5] At this point, students have gone through two semesters of introduction to programming, one semester of algorithms and data structures, and two semesters of logic. Given the prior knowledge necessary to follow the course, we do not think that it is possible to place this course earlier in the curriculum. On the other hand, the course is quite limited in time: It was initially devised to be taught in four ECTS units (European Credit Transfer System)[6] and is currently reduced to three ECTS units (i.e., just one-tenth of a semester's total effort). This results in an average weekly effort of 5 hours—two 1-hour lectures and 3 hours of individual practice and study. Table I sketches the structure of the course.

The contents for weeks 1–5 are commonplace in any traditional concurrency course, although the concepts (simultaneous execution, mutual exclusion and condition

[5]All the course contents (in Spanish) are available at http://babel.upm.es/teaching/concurrencia. English translation of selected materials—including a term project assignment with its solution and several multiple-choice tests—can be found at http://babel.upm.es/teaching/concurrency.

[6]ECTS units are defined as follows: 60 ECTS units is the total time (attending lectures + studying + doing homework, etc.) available in 1 year for a full-time student. One ECTS unit is roughly 27 hours of student work, and four ECTS is approximately one-seventh of a semester load.

Table I. Structure of the Course

| week | contents | milestones |
|---|---|---|
| 1 | Motivation and course intro. | |
| 2 | Concurrency in Java: process creation, provoking a race condition. | |
| 3 | Mutual exclusion by busy waiting; Lamport's protocol. | |
| 4 | Semaphore-based synchronization I. Simple synchronization schemes. | |
| 5 | Semaphores II. Less obvious schemes and limitations to implement condition synchronization. | |
| 6 | Shared resources: specification I. | |
| 7 | Shared resources: specification II. | |
| 8 | Review before first test. | First test. |
| 9 | Implementing shared resources in Java: synchronized methods. | |
| 10 | Shared resources with locks and conditions. | |
| 11 | Distributed systems and CSP. | |
| 12 | Implementing shared resources with the JCSP library. | |
| 13 | Review before shared memory project deadline. | Shared memory project due. |
| 14 | Review before second test. | Second test. |
| 15 | Review before JCSP project deadline. | JCSP project due. |
| 16 | Redesign workshop. | |

synchronization) are presented in an order that leads naturally to the introduction of shared resources in weeks 6–7.

Weeks 9–12 are devoted to the implementation of shared resources in Java by means of the coding patterns mentioned in Section 1 (synchronized methods, locks and conditions, and the *JCSP* library). The last two are used to develop the term project. A redesign workshop (see later in this section) takes place on the last week of the course.

The assessment activities include two test papers (on weeks 8 and 14), 10 weekly short exercises during the first half of the course intended to keep the students engaged with the subject, and a term project (turned in in two parts on weeks 13 and 15). The contribution of each of these parts to the final grade is as follows: written tests, 50%; term project, 40% (20% each part); short exercises, 10%.

As a result of the structure of the course and the limitations in time, the way in which it is taught had to be adapted to the environment.

—Students do not design a system from scratch. They start with an existing design provided by the instructors; they are asked to implement it and, later, to *redesign* it to adapt it to a change in the requirements.
—Students do not validate the design. The instructors ensure that the resource and the processes together have "good properties."
—Students receive tests to check their implementation. Ideally, by following the code generation patterns, working, correct executables can be mechanically derived, but of course bugs can appear when this process is manual.

## 3. PROFITING FROM A FORMAL SPECIFICATION

We have identified three main outputs obtained from the classroom usage of formal models: validation, code generation, and test generation.

### 3.1. Validated Design

The design of the concurrent system that the students receive should be consistent with the requirements. Undesirable effects, such as a deadlock in the system, should only be caused by an incorrect implementation. In this section, we present how the teacher validates its design and ensures that the specifications are consistent.

The semantics of shared resources is compatible with mature and well-understood verification tools. We usually translate specifications and processes into TLA+ [Lamport 2002] (a combination of the linear-time temporal logic "The Temporal Logic of Actions"

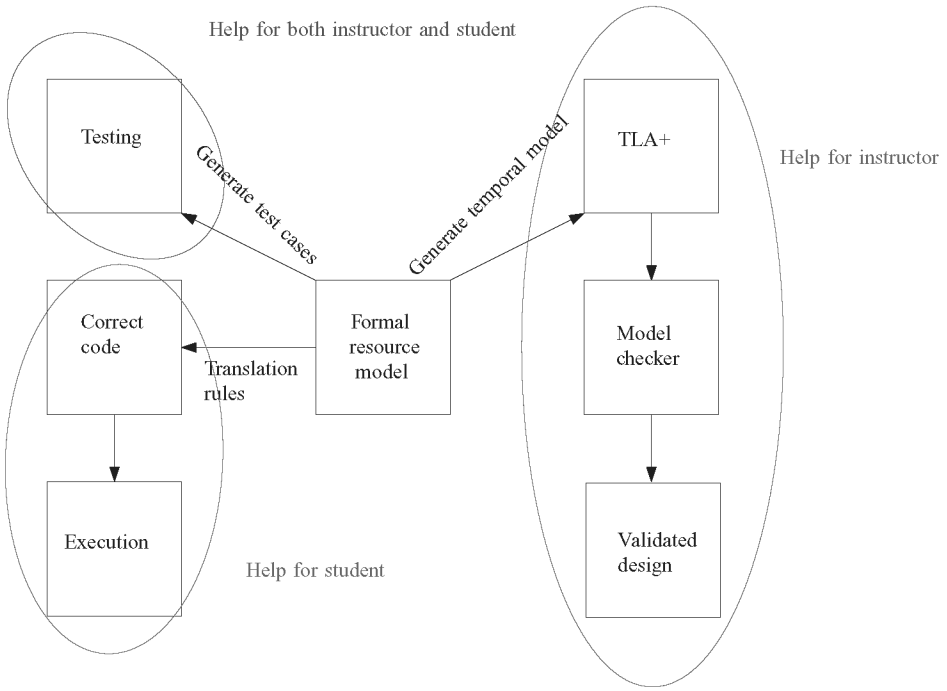


Fig. 4. Formal specification at the center of the development process.

[Lamport 1994] and Zermelo-Fränkel set theory), although sometimes we have used Uppaal [Behrmann et al. 2004] (a combination of timed automaton and timed computation tree logic). These allow us to automatically validate the design that the students will receive. In general, TLA+ is our preferred tool: In some sense, our shared resource specification language can be seen as syntactic sugar for TLA, so the translation is not very complicated, and TLC, the TLA+ model checker, is very powerful.

The design is validated in two different scenarios.

(1) *Validation of the shared resource in isolation.* Legal, calls to the shared resource can be made at any moment. The resource is translated into TLA+ together with a TLA formula, which states that legal calls can be performed at any moment. The TLC model checker is then used to check that the invariant (including type information) is not violated. If the model checker does not detect any problems in this scenario, then, regardless of which processes are used, no issues should appear. This is useful for generic resources whose design is independent from the calling processes (e.g., the selecting bounded buffer) (Figure 4). On the other hand, a negative result (i.e., if problems are detected) in a resource whose design depends highly on the processes is not definitive.

(2) *Validation of the system.* The logic of the processes is encoded into TLA+ and combined with the resource specification to explore only the interleavings that the real system should allow. In this case, the translation into TLA+ is more involved but still mechanizable. One advantage of this validation scenario is that stronger invariants can usually be proved. Of course, a positive outcome means that the resource design is reliable in the existing scenario.

Detailed information about the automatic translation of designs (shared resources and processes) into TLA+ can be found in Herranz et al. [2009].

```
class BBSel_Monitor implements BBSel {
   private List<Integer> data;
   private Monitor monitor;
   private Monitor.Cond[] condGet;
   private Monitor.Cond condPut;

   public BBSel_Monitor () {
      data = new LinkedList<Integer>();
      monitor = new Monitor();
      condPut = monitor.newCond();
      condGet = new Monitor.Cond[2];
      condGet[0] = monitor.newCond();
      condGet[1] = monitor.newCond();
   }

   public void put(int n)
   {
      monitor.enter();

      if (data.size() >= BBSel.MAX) {
         condPut.await();
      }

      data.add(n);

      condGet[data.get(0) % 2].signal();
   }

   public int get(int eo)
   {
      int result;

      monitor.enter();

      if (   data.size() == 0
          || data.get(0) % 2 != eo) {
         condGet[eo].await();
      }

      result = data.remove(0);

      if (   data.size() < MAX
          && condPut.waiting() > 0) {
         condPut.signal();
      }
      else if (data.size() > 0) {
         condGet[data.get(0) % 2].signal();
      }

      monitor.leave();

      return result;
   }
}
```

Fig. 5. Sample BBSel_Monitor.java implementation.

### 3.2. Mechanical Code Generation

The most tangible output of the model-driven approach is a piece of code implementing the resource's behavior. The code is obtained by means of coding patterns that transform the resource specification into executable code [Carro et al. 2004], possibly adding additional properties such as fairness.

Previous courses based on Ada 95 used two different coding idioms, one for *protected objects* and the other one for *rendez-vous*. In the current Java-based course, three idioms are presented to students. The first one, based on synchronized methods and notifyAll (), is quite straightforward and is presented as a way of quickly sketching a prototype implementation of the resource because enforcing liveness properties in that code is very complex. The other two idioms (locks and conditions and JCSP) are more elaborate, but in return, they give the programmer mechanisms to implement different liveness and priority policies. These are the ones used for the term project, and they result in quite different designs, as the former produces a monitor-like code with cascade signaling, whereas the latter implements the resource using a dedicated thread that acts as a server for the rest of the system, which allows for different message serving strategies, explicit management of channels, and so on. Due to space limitations, a detailed presentation of the translation cannot be given here, but we can illustrate the essence of the method with one possible implementation for the class BBSel (Figure 5).

The methods implementing the resource operations follow a common scheme, where serializability is ensured by enclosing the code in enter/leave blocks.[7] The body usually has three clearly separate segments: a condition synchronization part, the code that establishes the operation's POST, and the signaling protocol. The first and third parts

[7]Note that, while this may impact performance negatively if operations are large, we are in this course interested in concurrency and correctness, rather than in parallelism and performance.

obey a restriction: At most one await is executed in the former and at most one signal is executed in the latter. This allows to assert that the CPRE holds right after the blocking point, sinceMCL: (was "as long as") the responsibility for checking it falls on the signaler.[8] The previously described constraints are very helpful in giving the code a structure that is easy to follow and understand. Among the advantages of using this coding discipline, we can mention (i) efficiency, as there is no need to recompute CPREs right after awaking from an await, which could make a drastic difference in an example where traversing data structures can affect contention intolerably; and (ii) reasoning about the code, as there is at most one single point (await) where sequential reasoning is not applicable, and resumption ensures the operation's CPRE and the overall resource invariant.

There are different choices to implement the condition synchronization block. In the simplest case, the CPRE does not depend on any input parameter. Here, a condition queue per operation is enough. When some CPRE depends on input parameters of the operation, different strategies can be followed. When the conditions depend on data types with small ranges, such as in the bounded buffer with selection (Figure 5), simple schemes based on instantiating the CPRE and using an array of conditions can be used. In the general case, arbitrary data structures to map parameters to condition variables can be applied.

The last section of the code deals with signaling protocols. This is one of the trickiest parts, where intuition is often misleading. It may seem that there is a natural signaling ordering for get operations to signal calls to put operations that suspended and vice versa. However, a systematic study of which CPREs can hold after each POSTs reveals the need for other signaling possibilities that might have been overlooked.

The code that is needed to satisfactorily solve some assignments is admittedly complex and could hardly be correctly produced without applying model-driven development. However, their final structure, as generated by the code generation guidelines, is homogeneous across all operations with similar requirements and easy to understand. Therefore, students can use it as a guide for similar operations in other resources.

### 3.3. Automatic Generation of Tests

Testing is established as an industry-standard way to perform software quality assurance. Its relevance in education may be even higher, as it can be used in several directions. Using formal specifications to generate tests has been proposed elsewhere [Fernandez et al. 1997],[9] and we want to highlight the interest of this technique in the realm of education and teaching based on formal specifications.

The tester we generate replays a series of traces, and every trace represents one possible legal interleaving of the processes in the system. Every trace is executed sequentially by a single thread, which keeps track of the state of the processes it simulates, including local variables. The actual behavior of the operations, as implemented by the resource (which, in the usual case, has been coded by a student), is compared with the expected behavior according to the resource definition.

In particular, we check:

—whether a call to a resource operation that should proceed (i.e., which does not suspend) does so effectively and that the return value(s) are the right one(s);
—whether a call that should suspend does so effectively;

[8]Our students use our own implementation of locks & conditions with *priority* semantics.[Herranz-Nieva and Mariño 2011]

[9]Note that because we are dealing with reactive systems, we are not interested in generating test data (as in Offutt et al. [2003]).

—whether a previously suspended call that should resume after the completion of another call (which changes the resource state) actually resumes.

Traces (and, from them, the complete tester) are generated automatically by animating a representation of the processes and the resource. In our case, this representation was written in Prolog and executed by a generic driver (also written in Prolog), which explores by means of backtracking the possible states of the system up to a given depth, generates traces, and translates them into executable testers. The driver tries to generate interesting scenarios by exploiting a limited form of partial-order reduction and by not generating traces that have nondeterminism in the resumption.

A typical tester is some 80,000 lines long and executes between 500 and 1,000 different traces. When traces are executed, the tester keeps track of the performed calls and the input arguments. This makes it possible to write out a complete trace of the calls when a misbehavior is detected. Students can use this trace to find out what was wrong with their implementations.

## 4. DISCUSSION: PROS AND CONS OF A FORMAL APPROACH

As can be expected, the use of a formal approach to develop concurrent systems brings positive and negative traits when compared with other teaching styles, which, to our experience and knowledge, tend to focus on describing the concurrency capabilities of some language and show how solutions for some selected problems can be coded. While, of course, we do show solutions for classical problems, our aim gears more toward empowering the students with generic design tools. On the other hand, we do not intend to teach foundations of concurrency, and thus we take a pragmatic approach and rely on the intuitive understanding of what concurrent programming languages can do and how they behave. It is with this setting in mind that we try to candidly evaluate the advantages and disadvantages of our approach.

### 4.1. Formal Specifications

Using a formal specification gives the design an unambiguous semantics that replaces lengthy explanations to describe the behavior of the system and allows us to apply what basically is the same methodology while targeting different programming languages.

The most apparent disadvantage of our method lies on the difficulty for unskilled students to write and understand Z-based formal specifications.[10] To overcome this, a behavioral interface specification formalism [Hatcliff et al. 2012] could be used. Behavioral interface specification languages, such as the Java Modeling Language (JML) [Leavens et al. 1999], enable programmers to express invariants and guarantee/ensure annotations at code level, a less alien formalism for students. JML, for instance, could be easily extended to support annotations to specify that class instances are shared resources and that methods have CPREs.

A subtler problem lies in the combination of a formal approach and a mechanical derivation of code and tests. This causes some students not to perceive the dynamic nature of concurrent systems and the complexity of the interactions in them. Consequently, they are less likely to understand why the different synchronization patterns are implemented the way they are. This is clearly a weak point in our current course.

### 4.2. Validated Design

Because the design is the student's starting point, it is reasonable to expect that it is bug-free; otherwise, the work to be done by the student could be insurmountable. A

[10]Our students do not have special problems to understand specifications, but we acknowledge that our school may be an exception to this norm, as (first-order) logic has always been part of the freshman year.

correct implementation may face deadlock because the resource specification is wrong, and a student unaware of this may try to redo her code once and again to no avail. While in some cases it is evident that the resource is right, in other cases it is not so clear, and ensuring consistency and some basic properties (e.g., absence of deadlock) is of paramount importance. Our approach increases the confidence in the consistency of the designs, makes it also possible to attack problems of complexity higher than the ubiquitous bounded buffer, and avoids blaming the misbehavior of a student implementation on an incorrect initial design.

A drawback is that the translation to TLA+ and the checking of (finite) correctness is, as of now, too complex, not completely automated, and, overall, out of reach for the average undergrad student.

### 4.3. Mechanical Code Generation

Identifying and applying code patterns reduces the amount of mistakes made by students, and when bugs appear, these are restricted to specific code sections.

As a drawback, we leave less room for "programming as an art." We think that this is a pity; however, we also feel that the artistic part of programming should only be attacked when the "is the design correct?" box has been ticked. Still, there are two tasks that require the students to show their programming skills. First, in the translation of the sequential part of the specification into code, students have to interpret the specification and produce that code. Second, choosing the appropriate code generation pattern that covers a given situation needs understanding when these patterns are applicable.

### 4.4. Automatic Generation of Tests

Tests are designed to give detailed feedback of errors found and the trace of the calls that lead to these errors. For students, errors are revealed as a long and apparently uninformative list of calls, but following this series of calls forces students to understand how process interleavings can lead to unexpected results and to gain insight into how concurrent processes do behave.

An obvious negative aspect is the intrinsic incompleteness of the checks performed by the testers. This fact leads to a pair of effects. It is often not easy to make students fully aware of this lack of completeness, and implementations that pass the test (which can become a sort of staple argument) can be wrong. A more pernicious effect is that the ability students should have to exercise critical thinking on their own work and come up with test cases seems to end up impoverished. Nevertheless, this is a concern shared with any programming course.

## 5. ASSESSING THE ADVANTAGES OF THE METHOD

After walking through the positive and negative outcomes of our approach at a conceptual level, we will assess now how the use of the proposed method impacts the student's ability from two different perspectives: (i) ability to generate code that meets some specifications and (ii) how learning and applying the method impacts understanding concurrency.

The cohort we will use is made up by the students of the spring semester of 2011, which provided 63 data points. Although this is not a large population compared to all the students of concurrent programming, it contains all the students who were taught by the same instructors and had to complete the same tests, homework assignments, and term projects. Therefore, we have made the assumption that it is statistically significant because no data points in our population are outside the sample we have used. Given the high homogeneity of the data source, we think that its quality is good enough to draw reliable conclusions from it.

Table II. Quality of Code versus Use of Methodology

| | Bad | Good | Accum |
|---|---|---|---|
| Method | 0.13 / 8 | 0.24 / 15 | 0.37 / 23 |
| No Method | 0.27 / 17 | 0.37 / 23 | 0.63 / 40 |
| Accum | 0.40 / 25 | 0.60 / 38 | 1.00 / 63 |

Table III. Fraction of Students Having a Given Quality of Code in Each Use of Methodology Class

| | Bad | Good | Accum |
|---|---|---|---|
| Method | 0.35 | 0.65 | 1.00 |
| No Method | 0.43 | 0.57 | 1.00 |

### 5.1. Crafting Code

Our goal here is to find out whether the use of the methodology actually improves the quality of the code written by the students. We determined the code quality by submitting the code to an automatic tester, which executed the resource implementation in a series of increasingly complex scenarios that were automatically generated from the specification. The behavior of the resource, including process wake-up/resumption (Section 3.3), was compared with the behavior expected from the specification dictates.

We divided the students' homework into two classes (Bad or Good) depending on the number and complexity of tests they passed. Independently, they were classified depending on whether they have followed the methodology (Method or No Method).

Table II shows the contingency table for this analysis. Every cell contains the relative frequency of the elements represented by that cell and the total number of elements in that cell. Columns Bad and Good contain the data for the cases where the implementation was determined to perform correctly or not, independently of whether our method was followed. Rows Method and No Method, respectively, are related to whether the student has followed the course methodology.

The first conclusion that can be drawn is that (unfortunately) most students chose not to follow the methodology when doing the homework (Column Accum). This may come as a surprise, but in fact we did not require them to do so: A student who is able to write good code without following the methodology can pass the homework tests.[11] The relationship between "writing good code" and "following the methodology" cannot be directly deduced from that table, as the frequencies refer to students as a whole and are, therefore, biased by the number of students who chose to use or not to use the method. What we want to answer is "What is the influence of following or not following the method on generating good code?"

This is answered in Table III, where we have normalized rows Method and No Method so that the frequencies add up to one in each of them. From there, we can deduce that, given a student who followed the methodology, the probability that it had a good implementation (65% versus 35%) is higher than that of a student who did not follow the methodology (57% versus 43%)—and, of course, likewise for bad implementations.

However, a possibility is that good students (who should tend to write good programs) may naturally choose to apply the methodology; therefore, the superior performance of students applying the methodology is because they are intrinsically good. Therefore, we need to determine if the students who apply the methodology are predominantly good and if those who do not apply the methodology are, in general, performing under par.

To answer this question, Table IV displays the frequencies of students applying or not applying the methodology versus how good a student is. The measure of being a good student has been obtained by using grades from another separate but related course on Data Structures and Algorithms, which is delivered in the immediately previous semester. Note that using the grades from the Concurrent Programming course would

[11] Note that had we not accepted that, we could never have had points to separate students in the Method/No Method classes.

Table IV. Use of Methodology versus Student Quality

| | Lower 25% | Mid 50% | Upper 25% | Accum |
|---|---|---|---|---|
| Method | 0.13/8 | 0.10/6 | 0.14/9 | 0.37/23 |
| No Method | 0.11/7 | 0.41/26 | 0.11/7 | 0.63/40 |
| Accum | 0.24/15 | 0.51/32 | 0.25/16 | 1.00/63 |

Table V. Use of Methodology versus Student Quality, Row Normalized

| | Lower 25% | Mid 50% | Upper 25% |
|---|---|---|---|
| Method | 0.35 | 0.26 | 0.39 |
| No Method | 0.17 | 0.65 | 0.17 |

Table VI. Use of Methodology versus Student Quality, Column Normalized

| | Lower 25% | Mid 50% | Upper 25% |
|---|---|---|---|
| Method | 0.54 | 0.19 | 0.56 |
| No Method | 0.46 | 0.81 | 0.44 |

Table VII. Code Correctness Against Method Usage for Students in the Upper Range

| | Bad | Good |
|---|---|---|
| Method | 0.22 | 0.78 |
| No Method | 0.71 | 0.29 |

Table VIII. Code Correctness Against Method Usage for Students in the Mid Range

| | Bad | Good |
|---|---|---|
| Method | 0.17 | 0.83 |
| No Method | 0.35 | 0.65 |

Table IX. Code Correctness Against Method Usage for Students in the Lower Range

| | Bad | Good |
|---|---|---|
| Method | 0.63 | 0.38 |
| No Method | 0.43 | 0.57 |

give a biased view of what a good student is (i.e., we need an external oracle to tell us whether a student is good or bad). We have divided the students in three separate blocks: two containing the 25% of the students with the lower and upper grades (15 and 16 students, respectively) and a third containing the 50% of the students between these two extremes (32 students).

Let us study rows Method and No Method separately. If the methodology is applied in its major part by good students, we would expect the Cell (Method, Upper) to have the largest value of the row. Indeed, it is, but the difference with respect to the other cells in the same Method row is very small (see also Table V for a row-normalized table). Therefore, we can say with high confidence that the use of the methodology seems to be quite evenly spread across the range of the students, regardless of their performance. In particular, good and bad students do not show opposite tendencies.

If we now turn our attention into the No Method row, we could, again, expect that most bad students do not apply the methodology. Perhaps surprisingly, among those students who do not apply the methodology, bad students are the smallest fraction, and this fraction is, again, comparable to that of good students. The distribution in rows Method and No Method, therefore, confirms that good and bad students do not show any particular trend in their use of the methodology. Similar conclusions can be drawn from Table VI, which normalizes Table IV by columns: Students in the lower and upper part are roughly evenly divided in their use or not of the methodology, with a majority of them opting to use it.

Turning again to Table V, the distribution of cases in rows Method and No Method is completely opposite: There is a valley in the mid range in one case and a peak in the other case. Given this discrepancy, it seems relevant to study how using the methodology impacts the code of a student in each of these populations.

Tables VII to IX are row-normalized contingency tables of code correctness against use of the methodology for the students in each of the ranges. Students in the upper range show a clear correlation between good code and use of the method. Students in the mid range have a less clear bias, and students in the lower range seem to be the odd ones: Trying to apply the methodology, if anything, makes their code less likely to be correct. Although we currently do not have a good explanation, we conjecture that students in the low range have difficulties when deciding which strategy to apply and simply strive to have a working implementation.

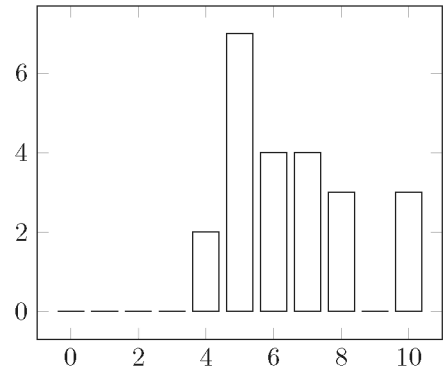


Fig. 6. Test grades for students who followed the methodology.

### 5.2. Understanding Concurrency

We have focused on the use of shared resources and the term project as drivers of the course. The term project is, in general, an exercise of synthesis, and capabilities related with analysis are mostly absent, with the exception of tasks related to the redesign of process schemes or shared resources. Therefore, additional assessment has to be provided to fill in the gaps not covered by the term project.

We implement these additional checks by using short weekly exercises and multiple-choice tests. Tests are designed to satisfy several quality indicators [Piontek 2008]. We aim at covering knowledge gaps such as the following.

—The term project often requires an uneven use of the different techniques for code generation. Some may be more demanding in terms of the translation of the synchronization preconditions, whereas others regarding the data structures needed to store pending operations.
—The lack of balance between synthesis and analysis.
—The need to prove that the rationale behind the model transformation is well understood so that the student can adapt it to a different situation (e.g., a different programming language).
—Very often, questions in our tests originate from mistakes revealed while doing supervision work of the students' homework. Misunderstandings are used to reinforce the weak parts of the course and to help future students.

We have measured whether applying the method has any effect on the understanding of the core concepts of concurrency. We have separated the grades obtained in the theoretical test by the students who followed/did not follow the method. The results are shown in Figures 6 and 7. Some conclusions can be drawn easily.

—The minimum grade is higher among the students who followed the method than among the group who not, which that is in line with our assumption that the methodology tends to avoid mistakes.
—The general shape of the distribution of those who followed the method leans more toward higher grades. Note that the graphics have a different population (as shown by the numbers in the vertical axis), but they are scaled to the same height, so the physical appearance gives a representation of the relative frequency of every case

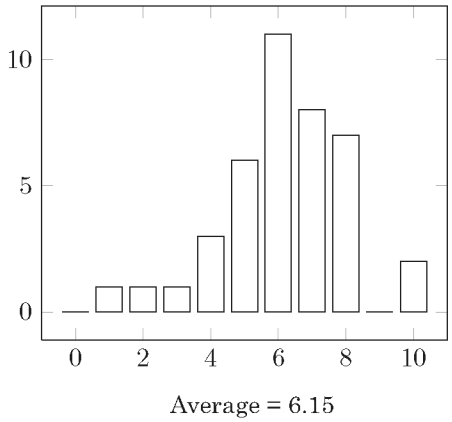


Fig. 7. Test grades for students who did not follow the methodology.

in its class. For example, the maximum grade was obtained much more often among the students who followed the method.
—The average grade is slightly higher among the students who followed the method.

These points lead us to assume that method followers, indeed, performed better on the tests as well.

## 6. CONCLUSIONS

In this article, we presented an undergraduate course on concurrency that makes extensive use of formal models. We offered details on the course structure, content, and methodology, as well as an analysis of advantages and risks of its adoption.

A summary of the main features of this article follows.

—*Formal models and semiformal design.* The students' starting point is a semiformal specification of the concurrent system they have to implement.
—*Validated design.* It is reasonable to expect that the design is bug-free. To ensure that it is, we translate the design to the language of an automatic verification tool.
—*Mechanical code generation.* Code patterns and idioms are taught to mechanically implement correct concurrent programs.
—*Automatic generation of tests and self-assessment.* Because coding errors can happen, we generate tests from specifications and make them available to allow the students to assess their implementation.

Our long-lasting impression, now backed up by statistical analysis, is that an approach to programming that starts with specifications and tries to deduce programs causes students to take longer to program fluently, but they tend to think before programming and, consequently, make fewer mistakes.

Perhaps, one of the merits of our proposal is its ordinariness: The fact that it has evolved swiftly over the years without pretending to be experimental, surviving changes in programming language, course structure, and student populations. We expect this work to be useful to instructors in at least two different ways. First, it can help those teachers already familiar with the model-driven approach to software construction and willing to incorporate it in the undergraduate curriculum to do it effectively and avoid some of its pitfalls. Second, it can present the model-driven approach as a feasible option to teachers concerned about some of the problems of learning

concurrency in the presence of programming language issues, a large number of students, or other challenges.

## ACKNOWLEDGMENT

The authors wish to thank the guest editors and the anonymous referees for their useful comments on earlier versions of this article.

## REFERENCES

BEHRMANN, G., DAVID, A., AND LARSEN, K. G. 2004. A tutorial on Uppaal. In *International School on Formal Methods for the Design of Computer, Communication, and Software Systems (Revised Lectures) (SFM-RT '04)*. Lecture Notes in Computer Science, vol. 3185, Springer, 200–237.

BEN-ARI, M. 2004. A suite of tools for teaching concurrency. *SIGCSE Bull. 36,* 3, 251–251.

BEN-ARI, M. 2009. Teaching concurrency and model checking. In *Proceedings of the 16th International SPIN Workshop on Model Checking of Software*. 6–11.

BRABRAND, C. 2008. Constructive alignment for teaching model-based design for concurrency. *Trans. Petri Nets Other Models Concurrency I*, Springer-Verlag, 1–18.

CARRO, M., HERRANZ, A., AND MARIÑO, J. 2009. Concurrent programming. http://ocw.upm.es/lenguajes-y-sistemas-informaticos/programacion-concurrente. In Spanish.

CARRO, M., MARIÑO, J., ÁNGEL HERRANZ, AND MORENO-NAVARRO, J. J. 2004. Teaching how to derive correct concurrent programs (from state-based specifications and code patterns). In *Proceedings of the Teaching Formal Methods, Co/LogNET/FME Symposium on (TFM '04)*. Lecture Notes in Computer Science, vol. 3294, Springer, 85–106.

FELDMAN, M. B. AND BACHUS, B. D. 1997. Concurrent programming can be introduced into the lower-level undergraduate curriculum. In *Proceedings of the 2nd Conference on Integrating Technology into Computer Science Education*. ACM, 77–79.

FERNANDEZ, J.-C., JARD, C., JÉRON, T., AND VIHO, C. 1997. An experiment in automatic generation of test suites for protocols with verification technology. *Sci. Comput. Program. 29,* 123–146.

GEHANI, N. H. 1993. Capsules: A shared memory access mechanism for concurrent C/C++. *IEEE Trans. Parallel Distrib. Syst. 4,* 7, 795–811.

GROSSMAN, D. AND ANDERSON, R. E. 2012. Introducing parallelism and concurrency in the data structures course. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*. ACM, New York, 505–510.

HATCLIFF, J., LEAVENS, G. T., LEINO, K. R. M., MÜLLER, P., AND PARKINSON, M. 2012. Behavioral interface specification languages. *ACM Comput. Surv. 44,* 3, 16:1–16:58.

HERRANZ, A., MARIÑO, J., CARRO, M., AND MORENO-NAVARRO, J. J. 2009. Modeling concurrent systems with shared resources. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems (FMICS '09)*. Lecture Notes in Computer Science, vol. 5825, Springer, 102–116.

HERRANZ-NIEVA, Á. AND MARIÑO, J. 2011. A verified implementation of priority monitors in Java. In *Proceedings of the 2nd International Conference on Formal Verification of Object-Oriented Software (FoVeOOS '11)*. Lecture Notes in Computer Science, vol. 7421, Springer, 160–177.

HOARE, C. A. R. 1974. Monitors, an operating system structuring concept. *Commun. ACM 17,* 10, 549–557.

LAMPORT, L. 1994. The temporal logic of actions. *ACM Trans. Program. Lang. Syst. 16,* 3, 872–923.

LAMPORT, L. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education, Inc.

LEAVENS, G. T., BAKER, A. L., AND RUBY, C. 1999. JML: A Notation for Detailed Design. *Behavioral Specifications of Businesses and Systems*, 175–188.

OFFUTT, A. J., LIU, S., ABDURAZIK, A., AND AMMANN, P. 2003. Generating test data from state-based specifications. *Softw. Test., Verif. Reliab. 13,* 1, 25–53.

PIONTEK, M. E. 2008. Best Practices for Designing and Grading Exams. Tech. rep. 24, Center for Research on Learning and Teaching, University of Michigan. http://www.crlt.umich.edu/publinks/CRLT_no24.pdf.

SADOWSKI, C., BALL, T., BISHOP, J., BURCKHARDT, S., GOPALAKRISHNAN, G., MAYO, J., MUSUVATHI, M., QADEER, S., AND TOUB, S. 2011. Practical parallel and concurrent programming. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)*. ACM, New York, 189–194.

SARASWAT, V. A. AND BRUCE, K. 2010. Curricula in concurrency and parallelism. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (SPLASH '10)*. ACM, New York, 281–282.

SPIVEY, J. M. 1992. *The Z Notation: A Reference Manual*. Prentice Hall, Hertfordshire, UK.

STEELE, JR., G. L. AND SARASWAT, V. A. 2009. Curricula for concurrency and parallelism. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, 703–704.

TAFT, T., DUFF, R., BRUKARDT, R., AND E.PLOEDEREDER, Eds. 2001. *Consolidated Ada Reference Manual. Language and Standard Libraries International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1*. Springer-Verlag.

WELCH, P. H., BROWN, N., MOORES, J., CHALMERS, K., AND SPUTH, B. H. C. 2007. Integrating and extending JCSP. In *Proceedings of the Communicating Process Architectures Conference (CPA '07)*. Concurrent Systems Engineering Series, vol. 65, IOS Press, 349–370.

YEAGER, D. P. 1991. Teaching concurrency in the programming languages course. In *Proceedings of the 22nd SIGCSE Technical Symposium on Computer Science Education*. ACM, New York, 155–161.