# Teaching How to Derive Correct Concurrent Programs from State-Based Specifications and Code Patterns*

Manuel Carro, Julio Mariño, Ángel Herranz, and Juan José Moreno-Navarro

Facultad de Informática
Universidad Politécnica de Madrid
28660 Boadilla del Monte, Madrid, SPAIN
{mcarro,jmarino,aherranz,jjmoreno}@fi.upm.es

**Abstract.** The fun of teaching and learning concurrent programming is sometimes darkened by the difficulty in getting concurrent programs to work right. In line with other programming subjects in our department, we advocate the use of formal specifications to state clearly how a concurrent program must behave, to reason about this behavior, and to be able to produce code from specifications in a semi-automatic fashion. We argue that a mild form of specification not only makes it possible to get programs running easier, but it also introduces students to a quite systematic way of approaching programming: reading and understanding specifications is seen as an unavoidable step in the programming process, as they are really the only place where the expected conduct of the system is described. By using formal techniques in these cases, where they are undoubtedly appropriate, we introduce formality without the need to resort to justifications with artificial or overly complicated examples.

**Keywords:** Concurrent Programming, Formal Specification, Code Generation, Safety, Liveness, Ada.

## 1 Introduction

At the Universidad Politécnica de Madrid (UPM), the intents of introducing rigorous development methods into mainstream Computer Science takes place in two fronts. In the research front, several groups work in the application of logic to the development of safe software through the design and implementation of multiparadigm programming languages such as Ciao [9] or Curry [6], or environments for the development of industrial software around formal specifications, such as SLAM [10]. But we have also a strong commitment on the *academic* front. In 1996, with the introduction of the current curriculum, programming courses were completely redesigned and formal specifications made an early appearance at the first year — *understanding* the problem at hand and *devising* a solution was given the same importance as the sheer fact of *coding* it.

* This research was partly supported by the Spanish MCYT project TIC2002-0055 and by EU projects FET IST-2001-33123 and FET IST-2001-38059.

**Teaching Programming and the Use of Formal Methods.** Classic approaches to programming courses in computing curricula tend to be *language centered*: learning is to some extent driven by the features available in some (often imperative) programming language (Ada, C, C++, Java...), and programming techniques follow from them. The typical argument is that students get a better motivation by having a *hands-on* experience as soon as possible. While this is true to some extent, there are also some problems. As the number of imperative constructions in many programming languages is large, they are usually presented in an informal manner and the corresponding classroom examples are just designed to practice the last piece of syntax. In most cases, students do not need to know all the idioms provided by a given programming language, and their use tends to obscure, rather than to illuminate, the principles behind algorithms, data, and program structures.

In this model, formal methods appear just as an *a posteriori* evaluation tool. Students have problems trying to interleave the program development process with these techniques, which are perceived as complicate, boring and useless. For instance, formal verification is introduced when the student is able to develop complex problems (e.g. a phone agenda), but it is used for almost toy examples (e.g. at most a QuickSort is verified). This is clearly disappointing.

We advocate an approach to the teaching of programming where a rigorous process is used from the very beginning as a design and development tool. But students will perceive these techniques as *useful*, and not as a necessary evil, only if a systematic way of developing programs is provided. The starting point is *problem specification*. A formal specification language is introduced in the first year. This language uses pre/post-condition pairs for describing the relationship between the input and output of every operation, using a language heavily based on first-order logic. We distinguish between *raw* specifications (with a distant connection to implementation) and *solved* specifications, refined in such a way that some generic method can be used to obtain a prototypical code from them. Most of the specification in the concurrent programming course can be considered in *solved form*, as an algorithm can be easily read in them.

Types are also introduced from the very beginning, as we consider modeling an essential activity of software development.[1] A basic toolkit and the ability to introduce new types suited to specific problems are also built into the specification language. In the second year the specification notation is extended to abstract data types and classes in order to cover the programming-in-the-large phase. Finally, in the third year the notation is further extended with annotations for concurrency.

We have been using a functional language (Haskell) as first target language, which allows our students to get familiar with programming and rigorous development more rapidly — and having more fun. Nevertheless, the use of a functional language is not compulsory, as the way to obtain imperative code from solutions is almost as easy as in the declarative case. In fact, at the end of the first semester imperative programming is introduced and Ada 95 [19] is used in most of the following courses.

All these ideas have been applied at our department over the last seven years, with satisfying results:

[1] By the way, something usually ignored even in textbooks where formal methods are used.

– Students attack problems from a systematic point of view, leaving *eureka* steps as a last resort.
– Students understand and acknowledge the role of a rigorous, careful methodology in the programming process.
– The average of marks is better, which (modulo interferences inherent to the teaching and examination process) makes it reasonable to think that more students are able to develop computer programs with a satisfactory degree of correctness.
– The knowledge of students seems to be more solid, as per the opinion of our colleagues from other subjects (Software Engineering, Databases, Artificial Intelligence, etc.).

**Organization of the Paper.** Next section discusses issues specific to teaching concurrency in the CS curriculum and how formal methods can be an improvement. Section 3 introduces the notation used to specify shared resources in an architecture-independent way. Section 4 (resp. 5) deals with the derivation of correct concurrent programs for the shared-memory (resp. message-passing) setting, starting from a shared resource specification. Section 6 discusses some related work in this area, and finally Sect. 7 summarizes successful aspects of our experience and points out shortcomings and possible improvements of this approach.

## 2 Teaching Concurrent Programming

Concurrent programming is not a prevalent topic in undergraduate curricula [7]. When it appears in a curriculum, it is usually approached from the perspective of teaching concurrency concepts and mechanisms, and seldom from the problem-solving perspective. Very often, known solutions for known problems are described, but not how solutions for new problems can be devised, which is left to the intuition of the students. This is aggravated by the inclusion of concurrency as part of subjects whose core belongs to operating systems and/or architecture,[2] and then concurrent programming has to serve the aims of the main subject. Sadly, this neglects the role of concurrency as a key aspect for the overall quality of almost every serious piece of software today – not just systems software – addressing issues such as usability, efficiency, dependability, etc.

For our students, concurrent programming is both an opportunity of having more fun – programs are suddenly interactive, and a new dimension appears: *time* – but also a challenging activity, as reasoning on the correctness (both partial and total) of a concurrent program may be rather involved. Discovering that previous ideas on how to debug a program are of little help when your application deadlocks or shows an unpredictable (and irreproducible) or unexpected behavior comes as a shock.

These difficulties make this subject an ideal vehicle for convincing students of the benefits of using methods which emphasize rigor in the development of high integrity software. In fact, years before 1996, concurrent programming was already the more logic-biased of our courses, and some kind of logic tables were used to aid the development of

[2] Even the Computing Curricula 2001 for Computer Science [16] includes concurrency into the operating systems area.

code for monitors. It took several years, however, to evolve these tables into a language for the specification of shared resources, separate the static and dynamic components of concurrent systems, and devise a development methodology.

There is a wealth of teaching material on concurrent languages and concepts that many authors have developed for years, and improving it is not an easy task. Therefore we do use introductory documentation on basic concepts [2,1] and on systems and language extensions. We try to make students aware of this material by devoting a sizable amount of time (the concurrent programming course spans over a semester) to going over it and classifying different language proposals in a taxonomy, while giving hints on how to adapt the development method we teach to languages other than Ada 95.[3] When dealing with, e.g., semaphores, this gives also precious insights about how compilers generate code for high-level concurrency proposals, but without the burden of having to deal with the whole compilation process.

After the introduction of these standard contents – properties and risks of concurrent programs (mutual exclusion, absence of deadlock, fairness, etc.); classic synchronization and communication mechanisms (from semaphores to monitors and CSP) – our students learn a development methodology that can be summarized in the following six stages:

1. Process identification.
2. Identification of inter-process interactions.
3. Defining the control flow of processes.
4. Process interaction definition.
5. Implementation/refinement of process interaction.
6. Analysis of the solution's properties (correctness, security and liveness).

Obviously, this is an iterative process. The main idea is that steps 1 to 5 should produce a *partially correct* solution, i.e. code that meets the safety requirements of the problem. Moreover, this code is generated in a semi-automatic way from the definition of interactions (stage no. 4.) Further iterations of stages 5 and 6 should only be used to enforce liveness and priority properties, or to improve the system's performance.

The other key idea is that the process is highly independent of the language and architecture used. This is clearly true of stages 1–4 and, for stage 5, specific code generation schemes (in our case for Ada 95) are provided, giving support to both shared-memory mechanisms (protected objects [19, Sect. 9.4]) and message-passing ones (rendez-vous [19, Sect. 9.7]). Some reasons for using Ada 95 as the development environment are: most programming courses in our University run with Ada so students do not need to learn a new language, Ada is not a toy language so serious programs can be implemented, Ada has mature built-in concurrency constructs (Ada Tasking Model) well suited to the code synthesis schemes, Ada is a very well designed programming language with high level abstraction mechanisms, and there are free tools available for several platforms.

Stage 1 (process identification) is done via an informal (but systematic) analysis of the interactions of the application, and abstract state machines are used in stage 6. We have left these issues out of the scope of this paper.

[3] Or CC-Modula [15], a variant of Modula developed in-house which featured semaphores, monitors, message passing, and conditional regions, and which was used for several years before Ada 95 was adopted.

**CADT** Resource Name
**OPERATIONS**
**ACTION** $\mathrm{Op}_1$: *Resource_Type[io]* $\times$ *Type$_1$[i]* $\times \ldots \times$ *Type$_n$[o]*
**ACTION** $\mathrm{Op}_2$: *Resource_Type[io]* $\times$ *Type$_1$[i]* $\times \ldots \times$ *Type$_n$[o]*
$\vdots$
**SEMANTICS**
**DOMAIN:**
**TYPE:** *Resource_Type* = ...
**INVARIANT:** $\forall r \in Resource\_Type \bullet I(r)$

**INITIAL**($r$): *A formula on $r$ specifying initial values for the resource*
**PRE:** $P(a_1, \ldots, a_n)$
***CPRE***: *This is an explanation of what the concurrency precondition means*
**CPRE:** $C(r, a_1, \ldots, a_n)$
$\mathbf{Op_1(r, a_1, \ldots, a_n)}$
***POST***: *This is an explanation of what the postcondition means*
**POST:** $Q(r, a_1, \ldots, a_n)$
$\vdots$

**Fig. 1.** Resource specification: a minimal template

One definite advantage of our approach is that it offers a general framework for analyzing problems where concurrency is mandatory, and a method to detect processes and resources (i.e., to wrap up the architecture of the system) and to specify and implement the system. Teaching design patterns in a language-independent fashion is also easier, since there is a language with which these can be expressed. To the best of our knowledge, the path we follow is novel in an undergraduate curriculum.

## 3 Notation and Logic Toolkit

We will now introduce shortly the formalism we use. This is necessary as it has been devised for the subjects taught at (and teaching style of) our institution. Although the core ideas are basically a simplification of well-known formal methods, such as VDM [14], some special constructions (regarding, e.g., concurrency) have been added. Following the classification in [20], our resource specifications are state-based, and this state is accessed only through a set of public operations. A template of a resource specification is shown in Fig. 1.

The development methodology we advocate relies strongly on the assumption that the system to design and implement can be expressed as a collection of processes which interact through shared resources (see Fig. 2) called **CADT** (for **C**oncurrent **A**bstract **D**ata **T**ype) in our notation. Introducing concurrency as an evolution of data types presents it as a generalization of data abstractions where emphasis is put on the interaction with the environment instead of on their internal organization and algorithms. As we will see

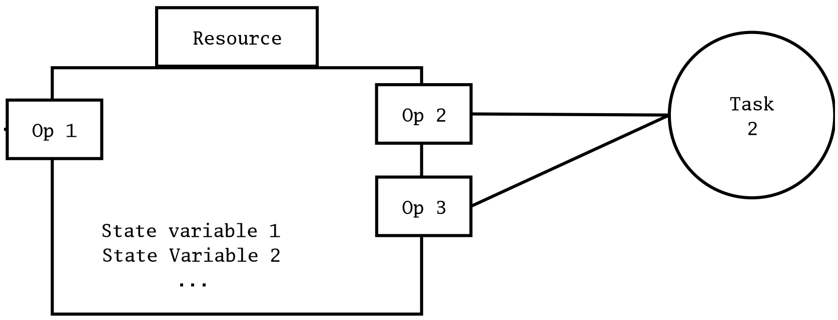


**Fig. 2.** Typical architecture of a concurrent program

later, it does not matter whether the final implementation is based on shared or distributed memory, as we have developed code generation schemes for both paradigms.

Unlike other approaches, our specification language does not aim at capturing the behavior of the processes, which are instead coded directly in the final programming language (or can even be derived using the methodology for sequential algorithms taught in other courses, which is out of the scope for this paper). In what follows we will give a brief account of the main characteristics of the specification language, leaving out some parts not needed for our purposes in this paper.

We will use as running example the specification and implementation of a multibuffer, in which processes can store and retrieve items from an encapsulated queue in series of $k$ elements, instead of just one by one. This example is simple to state, but at the same time it makes it possible to review quite a few different points: synchronization which depends on the input parameters, liveness properties which depend both on the interleaving of calls and on their runtime arguments, and different schedules based on priority requirements. The lack of need of a partial exclusion protocol, like the one in the readers/writers problems, is the only relevant missing characteristic.

We want to point out that the method we teach can easily go beyond this example to small prototypes of train barriers, video broadcast systems, robot interaction in industries, computer-controlled auctions, and a wide range of other non trivial cases, which we use as homework assignment and exam problems. See pointers to them at the end of Sect. 7. We consider that the average difficulty of these problems is high for an undergraduate course, and they certainly surpass that of the typical (but not less relevant) producers and consumers.

The specification language is strongly based on first-order logic, which is taught to most CS students at some point. Using it avoids introducing additional formalisms, reinforces the use of logic(s), often subject to misconceptions or poorly taught, and supports their role within computer science and related fields at several levels, from hardware design to program analysis and development [3].

### 3.1 Public Interface: Actions and Their Signatures

The **OPERATIONS** section defines the names and signatures of the public operations. Additionally, the input/output qualification of every argument can be optionally stated by marking them as *i* (input, immutable), *o* (output), or *io*; see the example below. We will show in Sect. 3.3 how changes to the arguments are expressed.

Unlike other approaches to specifying resources (e.g., [8]), the state is not directly available to the body of the specification, but it must be a formal parameter of every operation. We adopt the convention that this parameter is the leftmost one, and it has always input/output mode.[4]

**Example:** Operation names and signatures in the multibuffer

**CADT** MultiBuffer

**OPERATIONS**
**ACTION** Put: $Multi\_Buffer[io] \times Item\_Seq[i]$
**ACTION** Get: $Multi\_Buffer[io] \times Item\_Seq[o] \times \mathbb{N}[i]$

Note that **Put** does not receive the number of items to be stored — we assume that we want to deposit the whole sequence held in the second parameter. **Get**, on the other hand, receives the number of items to retrieve, but it could as well have received a sequence of the appropriate length.

### 3.2 Domain: Types and Invariants

We chose to have a relatively rich set of initial types which help in modeling different situations. This makes it possible to have short and understandable specifications, to factor out well-known issues related to data structures, and to focus on concurrency matters. We will now describe very briefly the available types and how invariants are written.

**Basic, Algebraic and Complex Types.** Basic types include booleans ($\mathbb{B}$), naturals ($\mathbb{N}$), integers ($\mathbb{Z}$), and real numbers ($\mathbb{R}$). We include also algebraic types to define enumeration, subranges, products, unions, and constructors. There is also syntax for sequences, sets, finite mappings, and to assign names to fields of algebraic constructors and components of product types. The availability of complex types helps to have specifications which are more readable and closer to what many programmers are used to. Note that many computer languages do not have builtin types for all of the above. Implementing them is a matter of another subject which escapes the present piece of work.[5]

We will here describe sequences very briefly, as they will be used in the rest of the paper. Sequences are a superset of lists and one-dimensional arrays. They represent

[4] We want to remark that this is not a key requirement. Adapting the specification to allow operations to refer to resource-wide variables does not affect greatly its syntax and semantics. We prefer, however, to keep it in a non object-oriented state for coherence with other subjects taught before at our school.

[5] But libraries are, of course, acceptable.

finite (but with no fixed length) indexed collections of elements. Assuming $s_1$ and $s_2$ are sequences and $i, j$ are integers, operations on sequences include finding their length ($\mathsf{Length}(s_1)$), accessing the $i$-th element ($s_1(i)$), accessing a subsequence ($s_1(i..j)$) and concatenating two sequences ($s_1+s_2$). Sequences are written with their elements between angle brackets, the empty sequence being $\langle\rangle$. A sequence of elements of type $T$ is declared as $\mathsf{Sequence}(T)$.

**Invariants.** The invariant is a formula which constrains the range of a type, aiming both at having only meaningful values and at specifying which states the resource must **not** evolve into: since the invariant does not have a notion of history, it can be used to state at most safety properties. The resource specification, and the processes accessing it, must ensure that banned states cannot be reached. For example, a type definition of a strictly increasing sequence follows:

**TYPE:** *Increasing* = $\mathsf{Sequence}(\mathbb{N})$
**INVARIANT:** $\forall s \in Increasing \bullet$
$(l = \mathsf{Length}(s) \wedge (l < 2 \vee \forall k, 1 \leq k \leq l-1 \bullet s(k) < s(k+1)))$

In the multibuffer example, a possible type definition is the following:

**Example:** Type definition for the multibuffer

**TYPE:** *Multi_Buffer* = $\mathsf{Sequence}(Data)$
*Item_Seq* = *Multi_Buffer*
**INVARIANT:** $\forall b \in Multi\_Buffer \bullet \mathsf{Length}(b) \leq MAX$

Note that the aim of the invariant here is just to set an upper limit to the size of the multibuffer. We have used the same data structure both for the multibuffer itself and for the parameters which store the data to be read and written. Since *Item_Seq* is of type *Multi_Buffer*, it is subject to the same constraints.

## 3.3 Specifying the Effect of Operations

Preconditions and postconditions are used to describe the changes operations make to the resource state, and when these operations can proceed. Both are first-order formulas which involve the resource and the arguments of the operations. For clarity reasons, we accept also natural language descriptions to back up (but not to replace) the logical ones.

**Synchronization.** We assume that resource operations proceed in mutual exclusion, but ensuring this is left to the final implementation, and is fairly easy to do in most languages (and automatic in Ada 95).

Condition synchronization is taken care of by means of concurrency preconditions (**CPRE**), which are evaluated against the state the resource has at the time of performing the (re)evaluation. A call whose **CPRE** is evaluated to *false* will block until a change in the resource makes it $true$, i.e., when some other process modifies the resource in the adequate direction. Since our design method assumes that the resource is the only means of inter-process communication, call parameters cannot be shared among processes (they should go into the resource in that case). This implies that their value can be changed only by the process *owning* them, and they cannot be updated while the call is suspended. A **CPRE** must therefore involve **always** the resource. From all calls to operations whose **CPRE**s evaluate to $true$, only one is allowed to proceed. We do not assume any fixed selection procedure — not even fairness.

**CPRE**s are intended to express safety conditions. Liveness properties might be dealt with at this level by adding state variables to the resource and enriching the preconditions. However, in most cases this makes specifications harder to read and hides safety properties. Besides, programming languages often have their own idioms to deal with liveness. Therefore, and as part of the methodology we teach, studying liveness properties is delayed until code with provable safety properties has been generated. This study is made more intuitive (but not less formal) with the help of a graph representing the states of the resource. From an educational point of view this is in line with a top-down development which aims at achieving correctness first.

Sequential preconditions (**PRE**) can be added to the operations. These are not aimed at producing code; rather, they are required to hold for the operation to be called safely, and ensuring this is responsibility of the caller. Naturally, **PRE**s should not reference the state of the resource, or races in its evaluation can appear. Having this distinction at the level of the specification language makes it clear which conditions stem from synchronization considerations, and which are necessary for data structure coherence.

**Example:** Condition synchronization in the multibuffer example

**PRE:** $quant \leq \mathsf{Length}(seq)$

**CPRE:** $\mathsf{Length}(mbuffer) \geq quant$

**Get(mbuffer, seq, quant)**

**POST:** $\ldots$

**CPRE:** $\mathsf{Length}(mbuffer + seq) \leq MAX$

**Put(mbuffer, seq)**

**POST:** $\ldots$

In this example, the synchronization uses the size of the multibuffer and the amount of data to be transferred. The **Get** operation uses the parameter *quant* to know how many items are to be withdrawn, and the **Put** operation uses the length of the sequence holding the data. Synchronization boils down to making sure that there are enough empty places/items in the multibuffer — calls to **Put/Get** would suspend otherwise. Additionally, the sequence passed to **Get** as parameter must be large enough to hold the required number of items; this is expressed by the **PRE** condition. Failure to meet that property can certainly cause malfunction.

**Updating Resources and Arguments.** Changes in the resource and in the actual call arguments are specified using per-operation postconditions (**POST**) which relate the state of the resource (and of the output variables) before and after the call. When **POST**s are executed, the **PRE** and **CPRE** of the operation and the invariant are assumed to hold.

Values before and after the operation are decorated with the superscripts "in" and "out", respectively.[6]

**Example:** State update in the multibuffer

We add the lacking postconditions to the previous piece of code:

**CPRE:** . . .
**Get(mbuffer, seq, quant)**
**POST:** $\mathit{mbuffer}^{\text{in}} = \mathit{seq}^{\text{out}}(1..\mathit{quant}) + \mathit{mbuffer}^{\text{out}}$

**CPRE:** . . .
**Put(mbuffer, seq)**
**POST:** $\mathit{mbuffer}^{\text{out}} = \mathit{mbuffer}^{\text{in}} + \mathit{seq}^{\text{in}}$

Since we had required that the length of the retrieved sequence be large enough to hold the number of items required, we just fill in a prefix of that sequence.

## 3.4 Process Code

The skeletons of two minimal processes (a consumer and a producer) which access the multibuffer using the shared variable M are shown below. The Data variables are assumed to be local to each process. When teaching we adopt directly the Ada 95 object style for task and protected object invocation. This slight syntax change does not surprise students at all.

**Example:** Skeletons of processes accessing the multibuffer

```
loop
    Get(Mb, Data);
    -- <Do something with Data>
end loop;
```

```
loop
    -- <Produce some Data>
    Put(Mb, Data);
end loop;
```

## 3.5 Other Goodies

The initial value of a resource can be expressed using a first-order formula. Specifying desirable concurrency or a necessary sequentiality among calls to resource operations is also possible. This is useful to perform a stepwise refinement towards a resource which does not require partial exclusion, or whose preconditions and postconditions can be fine tuned so that they do not perform unnecessary checks/suspensions.

## 4 Deriving Ada 95 Protected Objects

Concurrent programming based on shared memory is done via the *protected objects* mechanism of Ada 95. A protected object in Ada 95 is a kind of module (*package*, in Ada lingo) that guarantees mutual exclusion of public operations (*entries*, left column of Fig. 3). Protected objects can have also private entries which can be invoked only from inside the code of the same object. We will term them *delayed operations* because we will use them to split a public operation into several stages in order to suspend the caller

```
protected type Protected_Type is
entry Public_Op_1 (parameters);
  ...
private
   <Constant and variable declarations>
   <(resource state)>
   entry Private_Op_1 (parameters);
   ...
   <Constant and variable declarations>
   <(additional resource state)>
   ...
end Protected_Type;
```

```
protected body Protected_Type is
   entry Public_Op_1 (parameters)
   when condition is
   begin
      ...
   end Public_Op_1;
   ...
   entry Private_Op_1 (parameters)
   when  condition is
   begin
      ...
   end Private_Op_1;
   ...
end Protected_Type;
```

**Fig. 3.** Scheme of an Ada 95 protected object

task under some synchronization circumstances. They are shown in the right column of the code in Fig. 3.

Boolean conditions associated to every entry are called *guards* and are used to implement conditional synchronization. They are said to be *open* when they evaluate to *true*, and *closed* otherwise. Once a protected type has been defined and implemented, protected objects (instances of the protected type) can be declared, and operations on objects are invoked using an object oriented syntax:

```
PO_1, PO_2 : Protected_Type;
PO_1.Public_Op_i (actual parameters);
```

## 4.1 Dynamic Behavior of Protected Objects

This is a partial description of the behavior of a protected object when it has been invoked. The reader is referred to any of the several good books on Ada (e.g., [4,5]) for more precise details on protected objects and Ada 95 tasks.

When an operation is invoked on a protected object, the caller task must acquire exclusive read/write access first, suspending until any task with a lock on the object relinquishes it. Execution can proceed if the corresponding guard is open; the caller task is otherwise added to an *entry queue* and suspended until it is selected (or cancelled). Mutual exclusion (as it was required by the **CADT**s) is ensured by the protected object itself. This relieves the student from repeating once and again the same code pattern to achieve mutual exclusion, and leaves more time to focus on more complex concurrency matters.

Although conditional synchronization can often be directly left to the guards of each (public) entry, Ada 95 states (based on efficiency considerations) that guards can not refer to formal parameters of the operations, which is a clear handicap when **CPRE**s depend on them, as in the case of the multibuffer.

[6] Although this requirement is sometimes overlooked when the mode declaration in the signature is enough to disambiguate expressions.

```
entry Op_X (parameters)
when  CPRE is
   -- CPRE does not depend on parameters
begin -- CPRE holds here (just checked)
   -- PRE can be tested here
   <Op_X implementation>
   -- POST should hold here; it can be checked
end Op_X;
```

**Fig. 4.** Independence of the input data

Several approaches to overcome this limitation are found in the Ada literature [4], ranging from having multiple protected objects (when possible) to performing polling on the variables shared by the **CPRE** and the entry head. We, however, opt for a less "clever trick" type of approach which is applicable to any case.[7] This, in our opinion, furnishes the student with a (perhaps not very shiny) armor to fend off problems with, and which makes the implementation in itself not challenging at all. This leaves more time to focus on, e.g., design matters, which we have found to be one of the weaker points of our students.

### 4.2 Code Schemes for Condition Synchronization

In a first, general approach (see Sect. 4.4 for more interesting cases), each public operation in the resource is mapped onto a public entry of a protected object and, possibly, on one or more private entries. Distinguishing those cases is key to achieve a correct code; however, a syntactic analysis of the resource specification provides a safe approximation.

**Synchronization Independent of Input Data.** When the **CPRE** does not depend on the formal parameters of the operation (in a simplistic approach: when it does not involve them), the translation is straightforward, as shown in Fig. 4. Note that this is a very common case, found in many classical concurrency problems.

Note that in Fig. 4 runtime checking is added to the code. Although students are expected to be able to understand a specification well enough so as to generate correct code for postconditions and preconditions, we strongly advice to include these extra checks. They are usually easy to write — easier than crafting an entry body which implements constructively the postcondition — and they provide an additional support that the code is indeed correct. In a *production* stage (e.g., when the homework is handed in) these checks may be removed.

Remember also that **CADT**s do not allow to specify *side-effects*, i.e. change of state outside the resource or the actual parameters. According to our methodology, these should be placed in the processes' code.

**Synchronization Dependent on Input Data: Input Driven Approach.** When the **CPRE** uses formal parameters of the operation, the method we apply to overcome the

[7] The witty apprentice can always find in the design process a source of mind challenges.

limitations of Ada 95 resorts to using a more involved implementation which saves the state of the input parameters onto an enlarged object state, or which maps this state onto a larger program code. Both techniques use delayed entries.

In the latter case, new delayed entries (one for each of the instances of the **CPRE** obtained by instantiating the shared variables with all the values in their domain) are introduced. Let $\Phi$ be the formula corresponding to some **CPRE**, and let us suppose that $\Phi$ depends on the entry parameter $a_1 : D$ where $D = \{x_{11}, \ldots, x_{1n_1}\}$. The versions of $\Phi$ induced by the values of $a_1$ are:

$$\Phi[a_1 := x_{11}] \quad \ldots \quad \Phi[a_1 := x_{1n_1}]$$

where $\Phi[a_1 := x_{1i}]$ denotes $\Phi$ after substituting $a_1$ for $x_{1i}$. The process can be repeated if $\Phi$ depends on other parameters $a_2, \ldots, a_k$ (but we will assume that $k = 1$ and we will not use subscripts to name the variables). The resulting scheme is shown in Fig. 5. An advantage of this approach is that parameters do not need to be copied, and that the type $D$ does not matter — it can therefore be applied, in principle, to any program (when $D$ is finite). On the other hand, if $|D|$ is large, the number of delayed entries becomes impractical to be written manually.

```
entry Op_X (a,b)
-- CPRE depends on parameter a
when True is
begin
  case a is
    when x1 => requeue Delyd_Op_X_x1;
    ...
    when xn => requeue Delyd_Op_X_xn;
  end case;
end Op_X;
...
```

```
entry Delyd_Op_X_xi (a,b)
-- Private delayed entry for Op_X
when CPRE[a := xi] is
-- CPRE completely coded in the guard
begin
  -- CPRE holds
  <Op_X assuming a = xi>
  -- POST holds
  <Runtime assertions to check POST>
end Delyd_Op_X_xi;
```

**Fig. 5.** General scheme for parameter-dependent preconditions

Ada 95 has a code replication mechanism, termed *entry families* [19, Sec. 9.5.2 and 9.5.3] which makes it possible to write code parametric on scalar types (see the example in Sect. 4.3). While this solution works around replication problems in many cases, it has also some drawbacks: it cannot be applied to the case of complex or non-scalar types (e.g., floats), and using it when $|D|$ is very large may lead to low performance. We therefore recommend using it with care. Possible solutions to this problem are to abstract large domains, when possible, into a coarser data type (which needs some art and craft), or resort to solutions based on the *Task Driven Approach*, explained next.

**Synchronization Depends on Input Data: Task Driven Approach.** A solution to avoid a large number of replicated delayed entries is to move the indexing method from the types of the variables to the domain of tasks accessing the resource. In general, the number of tasks that may access the resource is smaller than the number of versions

```
entry Op_X (Caller : PID, a, b)             entry Delayed_Op_X(Caller : PID) (a, b)
-- CPRE depends on a. Only                  -- This is a private entry
-- one call with the same Caller            when CPRE[a := A_Copy(Caller)] is
when True is                                -- CPRE is completely coded
begin                                       begin  -- CPRE holds
   -- Save parameter in CPRE                   <Op_X assuming a = A_Copy(Caller)>
   -- into vectors A_Copy                      -- POST holds
   A_Copy(Caller):=a;                          <Runtime assertions to check POST>
   requeue Delayed_Op_X(Caller);            end Delayed_Op_X;
end Op_X;
```

**Fig. 6.** Scheme for the *Task Driven* approach (using entry families)

```
entry Op_X (a, b)                     entry Delayed_Op_X (a, b)
-- CPRE depends on a                  -- This is a private entry
when not Closed_X is                  when CPRE[a := A_Copy] is
begin                                 -- CPRE is completely coded
   -- Copy parameters                 begin
   A_Copy := a;                          -- CPRE holds
   Closed_X := True;                     <Op_X assuming a = A_Copy>
   requeue Delayed_Op_X;                 Closed_X := False;
end Op_X;                                -- POST holds
...                                      <Runtime assertions to check POST>
                                      end Delayed_Op_X;
```

**Fig. 7.** Scheme for the *One-at-a-time* approach

generated by the input driven approach, and in practice it is usually bound by some reasonable figure — and the resource can also simply put an upper limit on the number of requests that can be stored, pending to be reevaluated.

The method consists of introducing a delayed entry per possible process and adding a new parameter to identify that process. With this approach, at most one process will be queued in each delayed entry, and the parameters involved in the **CPRE** can be saved to the (augmented) resource state, indexed by the task identifier, and checked internally. If we let $PID$ be the type of task identifiers, the scheme we are proposing appears in Fig. 6.

**Synchronization Depends on Input Data: One-at-a-time.** Other techniques can be applied in order to reduce entry replication: for example, selecting a per-call identifier from a finite set in the code fragment between the public and the delayed entries, and assigning it to the call. The guard of the external entry will be closed iff all identifiers are in use. When this set is reduced to a single element, the resulting code is simple: arrays are not needed to save input parameters, and entry families are not necessary either (Fig. 7). Yet, it is able to cope with a wide variety of situations. As entries would not serve calls until the current suspended one has been finished, we have termed this scheme the "One-at-a-time" approach. While it restricts concurrency in the resource, the policy it implements is enough to ensure liveness in many a case.

## 4.3 Code for the Multibuffer Example

We will show here a direct derivation of the resource into protected objects using family entries indexed by the buffer size, as suggested previously. The specification of the resource is simple enough as to be mapped straightforwardly onto Ada 95 data structures. We want to note that this is often the case during the course, and algorithms and data structures have never been an issue in our experience. Also, in order to appreciate clearly concurrency issues, data have not been completely represented — we show only the length of the sequences of data.

**Example:** Multibuffer as a protected type

```
protected type MultiBuffer is
  entry Get (Items: in Quant_Range);
  entry Put (Items: in Quant_Range);
private
  Item_Counter: Buffer_Quantity := 0;
  entry Get_Fam(Quant_Range) (Items : in Quant_Range);
  entry Put_Fam(Quant_Range) (Items : in Quant_Range);
end MultiBuffer;

protected body MultiBuffer is
  entry Get (Items : in Quant_Range) when True is begin
    requeue Get_Fam(Items);
  end Get;
  entry Put (Items : in Quant_Range) when True is begin
    requeue Put_Fam(Items);
  end Put;
  entry Get_Fam (for Q in Quant_Range)
                (Items : in Quant_Range)
  when Q <= Item_Counter is begin
    Item_Counter := Item_Counter - Q;
  end Get_Fam;
  entry Put_Fam (for Q in Quant_Range)
                (Items : in Quant_Range)
  when Q <= Buffer_Size - Item_Counter is begin
    Item_Counter := Item_Counter + Q;
  end Put_Fam;
end MultiBuffer;
```

## 4.4 Complex Behavior and Fine Synchronization

In some situations it is impossible to implement a resource by using a straightforward translation, because mutual exclusion is inappropriate for some problems, or because a more fine grained control is necessary in order to implement *ad-hoc* scheduling patterns aimed at ensuring liveness properties.

**Partial Exclusion.** A simple design pattern is enough to cope with partial exclusion: the resource to be programmed has to include operations to signal when execution enters and exits the *partial exclusion zone*, similarly to the classical *Readers and Writers* problem. The resulting resource features full mutual exclusion, and can be treated as we have seen so far. The tasks must follow a protocol similar to:

```
Resource_Manager : Protected_Object;
...
Resource_Manager.Init_Op_X (actual parameters);
<Actual operation on the resource>
Resource_Manager.Finish_Op_X (actual parameters);
```

The scheme is identical to that used to implement mutual exclusion with semaphores, and it is subject to the same weaknesses — protocol violation would cause havoc. Therefore we do require that these operations are wrapped inside procedures (maybe into a package of their own) which ensures that the protocol is abode by.

**Finer Control on Suspensions and Resumptions.** Sometimes a fine-grain control is needed to decide exactly when suspended calls are to be resumed (because of, e.g., liveness conditions or performance considerations). Without entering in implementation details, in our experience, students used to end up mixing safety and liveness conditions before specifications were used extensively. Now, we expect for them to produce always safe code first, which as we have seen is easy to derive from the specification, and then to proceed to refine it in order to meet with efficiency/liveness conditions. In general, the code is transformed from guards such as

```
when Safeness_Condition is ...
```

to guards like

```
when (Safeness_Condition) and (Liveness_Condition) is ...
```

which will not violate safety, but in which the set of open guards is reduced. If only one guard is active at a time, the effect is precisely that of an explicit wakeup, which mimics the behavior of signals in semaphores or condition variables in the monitors — and which needs the same implementation techniques.

## 5 Deriving Message Passing Systems with Rendez-Vous

*Rendez-Vous* was the mechanism originally proposed for process communication and synchronization in Ada. It can be seen as a mixture of ideas from CSP and RPC. Expressiveness and semantics are those of an *alt* construct in CSP – synchronous communication; alternative, non-deterministic reception – but the syntax is more concise, resembling that of a remote procedure call.

These procedures are called *entries* (like in protected objects) and every input parameter hides a send from the client to the server, and every output parameter is a message back from the server to the client. This notation allows to express client-server solutions in a very elegant manner but, unfortunately, is not expressive enough to capture certain requirements, which motivated the introduction of protected objects and the *requeue* mechanism in Ada 95. Our approach here is to complement the rendez-vous mechanism with a sporadic use of a home-made implementation of channels and explicit message passing in order to overcome these limitations, thus obtaining a coherent method for distributed-memory concurrent applications in Ada.

```
task type Server_Type is
  entry Operation1 (parameters);
  ...
end Server_Type;
```

**Fig. 8.** Declaration of public services of a task

```
select
  when condition =>
    accept Op1 (parameters) do
    ...
    end;
    <Sentences outside the rendez-vous>
  or
  ...
end select
```

**Fig. 9.** Select loop in the body of a task

```
task body Server_Type is
  <Declaration of additional variables>
  <Initialization of the server state>
begin
  <Remaining initialization>
  loop
    select
      when CPRE1 =>
        accept Op1 (parameters) do
        ...
        end;
        <Sentences outside the rendez-vous>
      or
      ...
```

**Fig. 10.** Main loop for a server

Due to space limitations, and also to emphasize the pedagogical issues rather than the technical ones, our treatment of this part will be more schematic.[8]

Using the rendez-vous mechanism, the shared resource will be the property of a server process which will declare a number of public services to the client processes (Fig. 8). According to our method, these services will correspond, ideally, to the operations in the interface of a **CADT**. Client processes may invoke these operations using a syntax similar to that of protected objects. The code inside the server task is often a loop in which the alternative reception of requests takes place via the *select* construct (Fig. 9).

The semantics of the *select* is similar to that of the *alt* construct in CSP/Occam. Entries whose guards are evaluated to *false* are discarded. The remaining *accept* clauses are the services available, at this moment, to the clients. As in the original CSP proposal and similarly to protected objects, guards can only refer to the inner state of the server, never to formal parameters of the *accept* clause.

**Code Generation Schemes.** As in the protected object case, the simplest situation is a **CADT** where none of the **CPRE**s depend on input parameters. In this case the resource server will have an entry for each **CADT** operation and a main loop where services satisfying the **CPRE** will be made available to clients (Fig. 10).

If some **CPRE** depends on input parameters a two-stage blocking scheme – rather similar to that used with protected objects – will be used: there will be an *accept* clause in the *select* with the guard set to *true* which will be used to send the data needed to evaluate the **CPRE**, followed by a suspension of the client task until the **CPRE** holds so that the request is ready to be served. The evaluation of the pending **CPRE**s can take place outside the *select* loop (Fig. 11).

[8] At *http://babel.ls.fi.upm.es/publications/publications.html?keyword=concurrent*, the interested reader can find a full explanation including the rendez-vous code for the multibuffer example and the realization of channels.

```
task body Server_Type is
  <Declaration/initialization of the server's inner state>
begin
  <Remaining initialization>
  loop
    select
      when True =>
        accept OperationX (input parameters + reply channels) do
          <Store request>
        end;
      or
      ...
    end select
    while <there are pending requests to serve> loop
      <Extract (ReplyChannel, RequestData)>
      <Perform operation>
      ReplyChannel.Send(reply/ack);
    end loop;
  end loop
end Server_Type;
```

**Fig. 11.** Server loop for **CPRE**s dependent on input parameters

This two-stage blocking can be implemented via explicit message passing using a generic package *Channel* that provides a type for simple synchronous channels with *Send* and *Receive* operations. It is, thus, an explicit channel naming scheme, not present natively in Ada, but implemented using, in our case, protected objects. The client can be blocked by making it wait for a message from the server. The client task will therefore perform a call to the *entry* followed by an unconditional reception:

```
CReply : InstanceOfChannel.Channel_P;
...
Server_Type.OperationX (..., CReply);
CReply.Receive (reply/ack);
...
```

The answer from the server may be used to transmit output parameters of the **CADT** operation – when they exist – or just a mere acknowledgment. Observe that sending a reference to the reply channel allows the server to uniquely identify the client.[9] This is a clear counterpart of the process identifier strategy mentioned in Sect. 4.2.

Of course, mixed schemes, where some operations are synchronized on the guard and others use the two-stage blocking, are allowed.

**Explicit Signalling Using Channels.** The scheme presented above provides also a more straightforward and elegant mechanism for programming explicit wakeups than the one used with protected objects. Depending on

a) the data structures used to store pending requests, and
b) the selection criteria used to traverse them

[9] Which is rather usual in the client/server philosophy, e.g. by sending an `IP:port` address of the client to a web server.

different versions of a (partially correct) solution can be obtained fulfilling different liveness criteria.

Wakeups implemented via explicit sends from the server resemble more faithfully the ideas originally present in the *Signal* of old-time semaphores or the *Continue* in classic monitors. Remember that explicit wakeups could only be *simulated* when using protected objects by forcing all guards but one to be false. Explicit wakeups bring the following advantages:

**A more elegant code.** One problem with the protected objects code was that enforcing liveness/priority properties would often force to strengthen the entry guards, which, on one hand, led to losing the straight connection with the **CPRE**s and, on the other, would increase the risk of lacking concurrency or even deadlock. With the scheme introduced above, the liveness/priority logic is moved outside the *select* and the guards remain intact.

**Lower risk of starvation.** Another problem with explicit wakeups in protected objects (and also in monitors) is that waking up a set of waiting processes had to be done via a *cascade* of wake-ups, where each task finishing execution of an entry must establish the necessary conditions to wake up the following task, and so on. The logic implied by this mechanism is very error-prone, easily leading tasks to starvation if the cascade breaks.

With the server scheme, the loop following the *select* must ensure that the server will not enter the *select* while there are pending requests that could be served. This avoids the risk of new requests getting in the middle of the old ones, thus greatly reducing the risk of starvation.

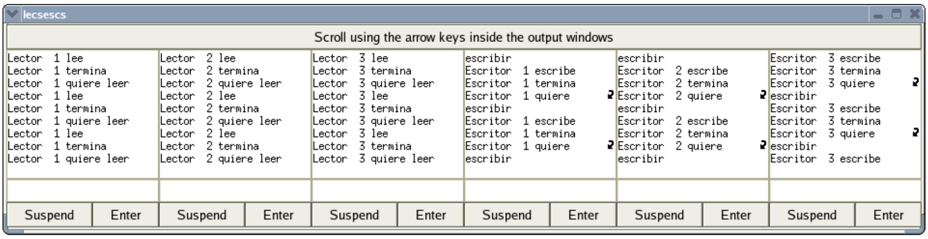
## 6 Related Work

To the best of our knowledge, there is not much work published on teaching concurrent programming as a self-contained subject — let alone teaching concurrent programming with the support of formal methods.

A pilot test reported in [7] supports the hypothesis that teaching concurrency to lower-level undergraduates increases significantly the ability to solve concurrency problems, and that concurrency concepts can be effectively learned at this level. Our own experience makes us agree with this view. Besides, we think that the use of a formal notation and the application of a rigorous development process helps in clarifying concepts with independence from the final implementation language and it really paves the way to having correct programs, even at undergraduate levels.

Undergraduate concurrency courses in the context of programming are also advocated in [13]. However, the approach of that paper is more biased toward parallelism than ours. We see parallelism as somewhat orthogonal to concurrency, and we tend to focus on interaction and expressiveness rather than on independence and performance.

Other pieces of work try to teach concurrency with the help of tools and environments which can simulate a variety of situations (see [17] and its references). This is indeed helpful to highlight peculiarities of concurrent programs, but from our point of view it does not help to directly improve problem-solving skills. That is what we aim at with a more formal approach.



```
Lector  1 lee
Lector  1 termina
Lector  1 quiere leer
Lector  1 lee
Lector  1 termina
Lector  1 quiere leer
Lector  1 lee
Lector  1 termina
Lector  1 quiere leer
```

```
Lector  2 lee
Lector  2 termina
Lector  2 quiere leer
Lector  2 lee
Lector  2 termina
Lector  2 quiere leer
Lector  2 lee
Lector  2 termina
Lector  2 quiere leer
```

```
Lector  3 lee
Lector  3 termina
Lector  3 quiere leer
Lector  3 lee
Lector  3 termina
Lector  3 quiere leer
Lector  3 lee
Lector  3 termina
Lector  3 quiere leer
```

```
escribir
Escritor  1 escribe
Escritor  1 termina
Escritor  1 quiere
escribir
Escritor  1 escribe
Escritor  1 termina
Escritor  1 quiere
escribir
```

```
escribir
Escritor  2 escribe
Escritor  2 termina
Escritor  2 quiere
escribir
Escritor  2 escribe
Escritor  2 termina
Escritor  2 quiere
escribir
```

```
Escritor  3 escribe
Escritor  3 termina
Escritor  3 quiere
escribir
Escritor  3 escribe
Escritor  3 termina
Escritor  3 quiere
escribir
Escritor  3 escribe
```

**Fig. 12.** Input / output of a *Readers / Writers* execution

In line with [21], we think that concurrent programming is of utmost importance. That piece of work also mentions the relationship concurrency / ADTs, but from a point of view different from ours: while our **CADT**s are concurrency-aware right from the beginning, that work seems to aim more at hiding concurrency than at exposing it.

Concurrency has also been animated with educational purposes, as in [11], which depicts dependencies among processes and semaphores. While we have not developed animations for Ada multitasking, we have built an Ada library which provides a subset of `Ada.Text_IO`, and which generates dynamically and user-transparently per-task input/output areas (Fig. 12). This is similar in spirit and motivations to [18], but with less system-oriented information, more user-transparent, and completely interactive (it runs in real time, in step with the main application).

## 7 Conclusion

Our students are taught concurrent programming according to the methodology herein presented. The subject is at undergraduate level, and delivered in the third year, after students have gone through several programming subjects in which a certain deal of specifications has been used. This makes the idea of reading and understanding a formal language not too strange for them.

We think that this is a success story: albeit the design of the concurrent system is by far the hardest task, when this is done students are able to develop almost mechanically Ada code for projects of fair complexity and with a high confidence on their reliability. Safety properties are guaranteed to hold, while liveness properties, when needed, have certainly to be developed with some care and on a case by case basis. We believe that being aware of the importance of keeping these properties is certainly a better investment than becoming an expert in, say, POSIX threads. These kind of abilities can be acquired later with comparatively little effort.

Due to space limitations, we have not detailed the last stages of the methodology which involve reasoning about the dynamic behaviour of resources and processes. This is done with the help of *labelled transition systems*. **CADT**s are still useful in this stage, as transitions are identified with invocations of **CADT** operations and states with (abstractions of) the state of the shared resources in the system. In other words, liveness issues can also be dealt with in an architecture-independent way.

Besides the translation schemes provided here, we have also developed, in previous stages of the curricula, similar translations for languages based on monitors [12] and on CSP. We have a similar translation scheme for Java, although probably not as clean as the ones we have presented here.

**Courseware Pointers.** Although the bulk of the information is in Spanish, we invite the reader to have a look at the web pages of our Concurrent Programming course at Universidad Politécnica de Madrid: `http://lml.ls.fi.upm.es/pc/`. Lecture notes, examples, homework assignments and test problems can be found at the subdirectories `apuntes`, `ejemplos`, `Anteriores/Examenes`, and `Anteriores/Practicas`.

## References

1. G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. In N. Gehani and A.D. McGettrick, editors, *Concurrent Programming*. Addison-Wesley, 1989.
2. Greg Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.
3. Maurice Bruynooghe and Kung-Kiu Lau, editors. *Program Development in Computational Logic: A Decade of Research Advances in Logic-Based Program Development*, volume 3049 of *Lecture Notes in Computer Science*. Springer, 2004.
4. Alan Burns and Andy Wellings. *Concurrency in Ada*. Cambridge University Press, 1998.
5. Norman H. Cohen. *Ada as a Second Language*. McGraw-Hill, 1995.
6. M. Hanus (ed.), H. Kuchen, and J.J. Moreno-Navarro et al. Curry: An integrated functional logic language. Technical report, RWTH Aachen, 2000.
7. Michael B. Feldman and Bruce D. Bachus. Concurrent programming can be introduced into the lower-level undergraduate curriculum. In *Proceedings of the 2nd conference on Integrating technology into computer science education*, pages 77–79. ACM Press, 1997.
8. Narain H. Gehani. Capsules: a shared memory access mechanism for Concurrent C/C++. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):795–811, 1993.
9. M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López García, and G. Puebla. The Ciao Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.
10. A. Herranz and J. J. Moreno. On the design of an object-oriented formal notation. In *Fourth Workshop on Rigorous Object Oriented Methods, ROOM 4*. King's College, London, March 2002.
11. C. William Higginbotham and Ralph Morelli. A system for teaching concurrent programming. In *Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, pages 309–316. ACM Press, 1991.
12. C.A.R. Hoare. Monitors, an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.
13. David Jackson. A Mini-Course on Concurrency. In *Twenty-second SIGCSE Technical Symposium on Computer Science Education*, pages 92–96. ACM Press, 1991.
14. Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1995.
15. R. Morales-Fernandez and J.J. Moreno-Navarro. CC-Modula: A Modula-2 Tool to Teach Concurrent Programming. In *ACM SIGCSE Bulletin*, volume 21, pages 19–25. ACM Press, September 1989.

16. The Joint Task Force on Computing Curricula IEEE-CS/ACM. Computing Curricula 2001. `http://www.computer.org/education/cc2001/`.
17. Yakov Persky and Mordechai Ben-Ari. Re-engineering a concurrency simulator. In *Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education*, pages 185–188. ACM Press, 1998.
18. Steven Robbins. Using remote logging for teaching concurrency. In *Procs. of the 34th SIGCSE Technical Symposium on Comp. Sci. Education*, pages 177–181. ACM Press, 2003.
19. T.S. Taft, R.A. Duff, R.L. Brukardt, and E.Ploedereder, editors. *Consolidated Ada Reference Manual. Language and Standard Libraries International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1.* Springer Verlag, 2001.
20. Axel van Lamsweerde. Formal Specification: a Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 147–159. ACM Press, 2000.
21. Dorian P. Yeager. Teaching concurrency in the programming languages course. In *Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, pages 155–161. ACM Press, 1991.