SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2006; **13**:1–323 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Teaching how to derive correct concurrent programs from state-based specifications

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

Universidad Politécnica de Madrid.
Facultad de Informática. Campus de Montegancedo s/n, 28660, Madrid, SPAIN.

SUMMARY

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 subjects in our department, we advocate the use of formal specifications to state clearly how a concurrent program must behave, to analyze and 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 the 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 behavior of the system is completely described. By using formal techniques in these cases, where it is undoubtedly appropriate, we introduce formality without the need to resort to justifications with artificial or overly complicated examples.

KEY WORDS: Concurrent Programming, Formal Specification, Code Generation, Safety, Liveness, Ada.

INTRODUCTION

At Universidad Politécnica de Madrid, the struggle to introduce formal 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 declarative programming languages (Ciao, Curry...) or environments for the development of industrial software around formal specifications (SLAM [**?**],). 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 — for the first time *knowing* the problem was given the same importance as *coding* it.

*Correspondence to: jmarino at fi.upm.es
Contract/grant sponsor: MCYT; contract/grant number: ghghjgj

Teaching Programming and the Use of Formal Methods

Classic approaches to programming courses in computing curricula tend to be *language centered*: learning is driven by the features in some (often imperative) programming language (Ada, C, Java...), and programming techniques follow. 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 is large, they are usually presented in an informal manner and the corresponding example is just designed to practice the last piece of syntax.

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 where formal techniques are used from the very beginning in the teaching of programming as a design and development tool. But students will perceive these techniques as *friendly* 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, at the function level, pre/post-condition pairs to describing the relationship between input and output, in a first-order logic setting. 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.

Types are also introduced from the very beginning, as we consider modeling an essential activity of software development[†]. 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 formalism 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 concurrent behavior.

We have been using a functional language (Haskell) as first language, which allows our students to get familiar with programming and formal methods 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 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:

- students attack any problem from a systematic point of view, leaving *eureka* steps as a last resort.
- students accept and understand the role of formal methods in the programming process,
- students obtain not only better marks but also better results, following the opinion of our colleagues from other subjects (Software Engineering, Databases, Artificial Intelligence, etc.)

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

Teaching Concurrent Programming

For a Computer Science student, 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 and discovering that previous ideas on how to debug a program are of little help when your application deadlocks or shows an unpredictable behavior.

These difficulties make this subject an ideal vehicle for convincing students of the benefits of using formal methods 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 monitors code. It took several years, however, to evolve these tables into a formal language for the specification of shared resources, separate the static and dynamic components of concurrent systems and devise a development methodology.

In addition to the standard contents of CP courses – 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 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 and, moreover, this code is generated in a semi-automatic way from the formal definition of interactions (stage no. 4.) Further iterations of stages 5 and 6 should only be used to enforce liveness and priority properties.

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 [**?**, section 9.4]) and message-passing ones (rendez-vous [**?**, section 9.7]). Some reasons for using Ada 95 as the development environment are: most programming courses in our University run with Ada so students does 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 design programming language with high level abstraction mechanisms, and there are available free tools under 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.

CADT Resource Name
OPERATIONS
ACTION Op_1: *Resource_Type[io]* $\times$ *Type*$_1$*[i]* $\times \ldots \times$ *Type*$_n$*[o]*
ACTION 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: $C(r, a_1, \ldots, a_n)$
Op$_1$**(r, a**$_1$**, . . . , a**$_n$**)**
***POST*:** *This is an explanation of what the postcondition means*
POST: $Q(r, a_1, \ldots, a_n)$
$\vdots$

Figure 1. Resource specification: a minimal template

Organization of the Paper

Next section introduces the notation used to specify shared resources in an architecture-independent way. Section (resp.) deals with the derivation of correct concurrent programs for the shared-memory (resp. message-passing) setting, starting from a shared resource specification. Section 1 summarizes, highlights successful aspects of our experience and points out shortcomings and possible improvements of this approach. Due to space constraints, some detailed descriptions appear in appendices at the end of the paper.

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 [**?**], some special constructions (regarding, e.g., concurrency) have been added. Following the classification in [**?**], 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.

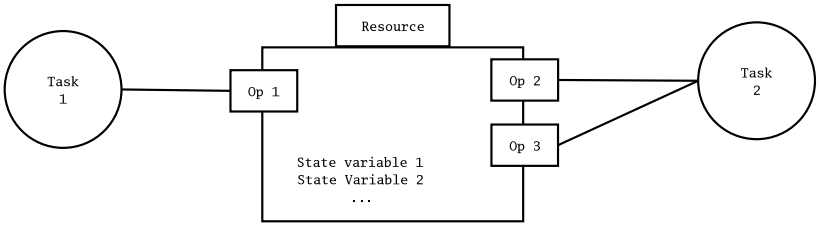

Figure 2. Typical architecture of a concurrent program

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 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. 1. 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 [**?**].

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. how changes to the arguments are expressed.

Unlike other approaches to specifying resources (e.g., [**?**]), 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.[‡]

Example: Operation names and signatures in the multibuffer

CADT MultiBuffer

OPERATIONS
ACTION Put: *Multi_Buffer[io]* × *Item_Seq[i]*
ACTION Get: *Multi_Buffer[io]* × *Item_Seq[o]* × 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.

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 (𝔹), naturals (ℕ), integers (ℤ), and real numbers (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

[‡]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.

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.[§]

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

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

Example: Type definition for the multibuffer

TYPE: $\mathit{Multi\_Buffer} = \mathsf{Sequence}(\mathit{Data})$
$\mathit{Item\_Seq} = \mathit{Multi\_Buffer}$

INVARIANT: $\forall b \in \mathit{Multi\_Buffer} \bullet \mathsf{Length}(b) \leq \mathit{MAX}$

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.

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.

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

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.

CPREs 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: ...

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

Put(mbuffer, seq)

POST: ...

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.[¶]

Example: State update in the multibuffer

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

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

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

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.

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;
```

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.

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

```
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;
```

Figure 3. Scheme of an Ada 95 protected object

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 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*);

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., [**?**, **?**]) 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.

```
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;
```

Figure 4. Independence of the input data

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.

Several approaches to overcome this limitation are found in the Ada literature [**?**], 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.[‖] 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.

Code Schemes for Condition Synchronization

In a first, general approach (see Sect. 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

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

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 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 Φ be the formula corresponding to some **CPRE**, and let us suppose that Φ depends on the entry parameter $a_1 : D$ where $D = \{x_{11}, \ldots, x_{1n_1}\}$. The versions of Φ 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 Φ after substituting a_1 for x_{1i}. The process can be repeated if Φ 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.

Ada 95 has a code replication mechanism, termed *entry families* [**?**, Sec. 9.5.2 and 9.5.3] which makes it possible to write code parametric on scalar types (see the example in Sect.). 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 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.

```
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;
```

Figure 5. General scheme for parameter-dependent preconditions

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.

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.

```
entry Op_X (Caller : PID, a, b)
-- CPRE depends on a. Only
-- one call with the same Caller
when True is
begin
  -- Save parameter in CPRE}
  -- into vectors A_Copy}
  A_Copy(Caller):=a;
  requeue Delayed_Op_X(Caller)};
end Op_X;

entry Delayed_Op_X(Caller : PID) (a, b)
-- This is a private entry
when CPRE[a := A_Copy(Caller)] is
-- CPRE is completely coded
begin -- CPRE holds
  <Op_X assuming a = A_Copy(Caller)>}
  -- POST holds
  <Runtime assertions to check POST>}
end Delayed_Op_X;
```

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

```
entry Op_X (a, b)
    -- CPRE depends on a
when not Closed_X is
begin
    -- Copy parameters
  A_Copy := a;
  Closed_X := True;
  requeue Delayed_Op_X;
end Op_X;
```

```
entry Delayed_Op_X (a, b)
    -- This is a private entry
whenCPRE[a := A_Copy] is
    -- CPRE is completely coded
begin
    CPRE holds
  <Op_X assuming a = A_Copy>
  Closed_X := False;
    -- POST holds
  <Runtime assertions to check POST>
end Delayed_Op_X;
```

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

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;
```

Complex Behavior & 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.

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 will be to complement the rendez-vous mechanism with an sporadic use of channels and explicit message passing in order to overcome these limitations, thus obtaining a coherent method for distributed-memory concurrent applications in Ada.

Ada's *Rendez-Vous*

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:

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

According to our method, these services will be the operations in the interface of a CADT. Client processes may invoke these operations using a syntax similar to that of protected objects:

```
Server_Task.OperationX (parameters);
```

**Protected objects did not appear until Ada 95.

The code inside the server task is often a loop in which the alternative reception of requests takes place, via the *select* construct:

```
select
    when condition =>
        accept Operation1 (parameters) do
            ...
        end;
        <sentences outside the rendez-vous>
  or
  ...
end select
```

The semantics of the *select* is similar to that of the *alt* construct in CSP/Occam. Guards are evaluated in first place and those that evaluate to *false* are discarded. The *accept* clauses whose guard evaluated to *true* are the available services, at this moment, to the clients.

- If, when the server task reaches the *select*, some client is waiting on a call to one of the available services, one of these calls will be selected for execution.[††]
- On the other hand, if no client has communicated its interest in any of the available services yet, the server will block waiting for the first request to arrive, which will be served immediately.

When one of the services is selected, only its corresponding accept clause is executed. There are two different parts in this code: the rendez-vous itself, from the "`do`" until its closing "`end`" and the remaining sentences of the clause until `or` or the `end select`.

The rendez-vous is the synchronization point among client and server:[‡‡] the client whose request is being served blocks until the server reaches the end of the rendez-vous. The remaining sentences can already be executed concurrently with the client code which follows the entry call — note that this makes a difference with entries in a protected object.

It is crucial to take this behavior into account: slow operations placed inside the rendez-vous code will decrease the overall concurrency. Blocking operations called from inside the rendez-vous section will also block any client waiting on in. As rule of thumb, this kind of operations should be placed outside the rendez-vous area.

The rendez-vous is also the scope for formal parameters in the *accept* clause. This makes sense, as it forbids the programmer modifying the output parameters beyond the point of synchronization with the client — which could be already modifying or destroying them concurrently.

As in the original CSP proposal and similarly to protected objects, guards can only refer to the inner state of the server, never formal parameters of the *accept* clause.

[††]No fixed policy is specified in [**?**].
[‡‡]Hence the name.

Code Generation Schemes

The simplest situation is a CADT where none of the CPREs depend on input parameters. In this case the resource server will have an entry per CADT operation and a main loop where services satisfying the CPRE will be made available to clients:

```
task body Server_Type is
    <declaration/initialization of the server's state>
begin
    <remaining initialization>
    loop
        select
            when CPRE1 =>
                accept Operation1 (parameters) do
                    ...
                end;
                <sentences outside the rendez-vous>
        or
        ...
```

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 blocking of the client until the CPRE holds so that the request is ready to be served.

This two-stage blocking will 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.

Blocking of the client can be achieved by either making it wait on a *Send* (to the server) or an acknowledgment message (from the server) depending of the situation. A scheme in which the second possibility is taken is shown below:

```
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;
```

The client task will 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 identify the client.

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

Example: The multibuffer, client-server version

```
task type MultiBuffer is
   entry Get (Num_Items       : in     Buffer_Quantity;
              The_Get_Channel : in out Channel_P);
   entry Put (Num_Items       : in     Buffer_Quantity;
              The_Put_Channel : in out Channel_P);
end MultiBuffer;
task body MultiBuffer is
   Item_Counter: Buffer_Quantity := 0;
   Delayed_Gets: Queue_Reqs.Queue := Make_Empty_Queue;
   Delayed_Puts: Queue_Reqs.Queue := Make_Empty_Queue;
   Items_To_Move: Natural;
   Delayed_Req: Request;
   Done: Boolean;
begin
   loop
      select
         when True =>
            accept Get (Num_Items       : in  Buffer_Quantity;
                        The_Get_Channel : in out Channel_P)
            do
               Insert (Delayed_Gets, (The_Get_Channel, Num_Items));
            end Get;
      or
         when True =>
            accept Put (Num_Items       : in     Buffer_Quantity;
                        The_Put_Channel : in out Channel_P)
            do
               Insert (Delayed_Puts, (The_Put_Channel, Num_Items));
            end Put;
      end select;
      Done := False;
      while not Is_Empty(Delayed_Puts) and
```

```
            not Is_Empty(Delayed_Gets) and not (Done) loop
         First(Delayed_Puts, Delayed_Req);
         if Delayed_Req.Amount <= Buffer_Size - Item_Counter then
            Delete(Delayed_Puts);
            Item_Counter := Item_Counter + Delayed_Req.Amount;
            Put_line(Buffer_Quantity'Image(Delayed_Req.Amount));
            Delayed_Req.Channel_Req.Send(Void_Message);
            Put_line(Buffer_Quantity'Image(Delayed_Req.Amount));
         else
            Done := True;
         end if;
         First(Delayed_Gets, Delayed_Req);
         if Delayed_Req.Amount <= Item_Counter then
            Delete(Delayed_Gets);
            Item_Counter := Item_Counter - Delayed_Req.Amount;
            Delayed_Req.Channel_Req.Send(Void_Message);
            Done := False;
         end if;
      end loop;
   end loop;
end MultiBuffer;
```

Observe the use of channels and how pending requests are stored in queues.

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

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 entries' guards, which, on one hand, led to losing the straight connection with the CPREs 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 processes waiting had to be done via a *cascade* of wake-ups, where each task finishing execution of an entry must establish the conditions necessary to wake

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, greatly reducing the risk of starvation.

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 [**?**] 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 [**?**]. 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 [**?**] 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.

In line with [**?**], 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 [**?**], 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. 8). This is similar in spirit and motivations to [**?**], but with less system-oriented information, more user-transparent, and completely interactive (it runs in real time, in step with the main application).

1. CONCLUSION

Our students are taught concurrent programming according to the formal 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 formal specification has been

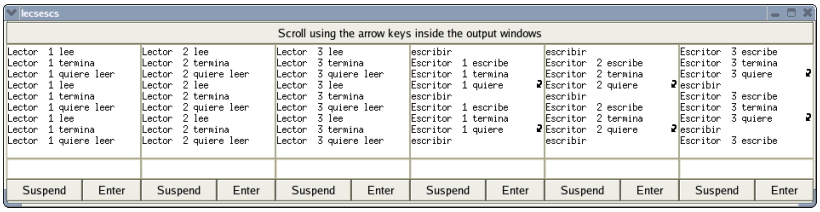

Figure 8. Input / output of a *Readers / Writers* execution

used. This makes the idea of reading and understanding a formal language not as *alien* as one may think at first.

We sincerely 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 go hold by the development method, 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, using POSIX threads. These, and similar, abilities can be later acquired with comparatively little effort.

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

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

Specification and Implementation of the `Channel` Package

`channel.ads`

```
generic
   type TMessage is private;
package Channel is


protected type Channel is
   entry Send (Data : in TMessage);
   entry Receive (Data : out TMessage);
private
   Item   : TMessage;
   Data_OK: Boolean := False;
```

```
   Waiting2Send: Boolean := False;
   entry Send2 (Data: in TMessage);
end Channel;

type Channel_P is access Channel;
procedure Destroy_Channel(C: in out Channel_P);
end Channel;
```

channel.adb

```
with Ada.Unchecked_Deallocation;

package body Channel is

   protected body Channel is
      entry Send (Data : in TMessage)
      when not Data_OK and
           not Waiting2Send
      is
      begin
         Item := Data;
         Data_OK := True;
         Waiting2Send := True;
         requeue Send2;
      end Send;

      entry Send2 (Data : in TMessage)
      when not Data_OK is
      begin
         Waiting2Send:= False;
      end Send2;

      entry Receive (Data : out TMessage)
      when Data_OK is
      begin
         Data := Item;
         Data_OK:= False;
      end Receive;

   end Channel;

   procedure Free_Memory is
     new Ada.Unchecked_Deallocation (Object => Channel,
                                     Name => Channel_P);

   procedure Destroy_Channel(C: in out Channel_P) is
   begin
      Free_Memory(C);
      C:= null;
   end Destroy_Channel;
end Channel;
```