Direct semantics of concurrent languages in the SMoLCS approach

by Egidio Astesiano
Gianna Reggio

For years providing syntax-directed methods for the formal definition of concurrent languages has proved to be a challenging task. Problems are even more difficult if a language has some of the typical Ada features, such as strong interference between sequential and concurrent aspects, parameterized semantics, complex data structure, and finally an extremely large size. We have developed an approach, the SMoLCS approach, which extends the denotational method to handle concurrent languages and also provides a solution to the above problems. Indeed, our method has been adopted for the formal definition of full Ada within the related EEC project. Here we illustrate the basic principles of the approach, following the so-called direct semantics style used for Ada with the help of a toy language as a running example.

1. Motivation and content

Considerable effort on the formal modeling of concurrency has now given us the possibility of devising nice semantics, at least for simple, well-structured concurrent languages.

However, languages such as Ada® pose a number of problems which cannot be solved just by relying on a simple model for concurrency. Indeed, its large size requires modular specification techniques; the strong interference between concurrent and sequential aspects, hidden by a syntax which is mainly aimed at static checks, must be resolved by defining a precise and possibly abstract underlying concurrent model and then by connecting the syntax to that model; moreover, the overall semantics of a program or of a fragment of it not only can vary depending on some parameters, which can be complex specifications, but is not at all fixed under any commonly accepted meaning (it is fixed in a rather sophisticated sense). Most of the problems have been brought to light and confirmed in practice by some early attempts to formalize the semantics of Ada (see [1] for problems and references).

Within this context, and motivated by a concrete large project (see [1–3]), the European Community Project on the Draft Formal Definition of Ada, we have developed a formal method for solving the problems mentioned above. For a long time it has been recognized that syntax-directed (or compositional) semantics is a first step toward modular definition; moreover, the denotational style advocated by Strachey and Scott (see [4, 5]), and also supported by the VDM work [6], has now become quite well understood and accepted. Hence, we have tried to extend the denotational

This work was partially funded by CNR and MPI (40%), and was developed in connection with the work of the CRAI–Genoa group (E. Astesiano, A. Giovini, F. Mazzanti, G. Reggio, and E. Zucca) for the EEC project "The draft formal definition of Ada" (DDC–CRAI–IEI–Universities of Genoa, Pisa, and Lyngby).

Ada® is a registered trademark of the U.S. Government (Ada Joint Program Office).

approach from sequential to concurrent languages. In a first attempt we used the so-called continuation style; a short outline of this approach is given in [7]. Then we succeeded in extending our approach to the so-called direct-style semantics, which was then used in the project for the formal definition of full Ada.

In this paper we present our approach to direct denotational semantics for concurrent languages, which has now become known as the SMoLCS approach. There are a few basic ideas underlying our approach, which we now briefly outline.

Our method involves two steps. First, a syntax-directed translation is performed, producing an intermediate language which is suitable for representing processes and their concurrent interactions. In this step the interference between concurrent and sequential aspects is resolved by making explicit what is truly a sequential activity and what is concurrent. Then the semantics of the intermediate language is provided.

One of the major novelties of SMoLCS is that we have the possibility of defining the intermediate language and its semantics following a parameterized schema, which can accommodate user-defined language constructs and their semantics. Indeed, the specification of the intermediate language and of its semantics is just the specification, following a predefined schema, of a concurrent system corresponding to the underlying concurrent structure of the source language. This is a most important feature, because it allows us to keep, as high as permitted by the language, the level of the primitives for handling concurrency, without translating them into a low-level fixed language.

A second important novelty consists in considering a concurrent process or system to be just a data type defined by a particular abstract algebraic specification called an algebraic transition system. The advantage of this technique is that it makes possible the handling of data types and processes in a uniform framework, using classical methods for guaranteeing abstraction, modularity, and parameterization.

Moreover, we can adopt some well-known techniques for defining different kinds of semantics and even parameterized semantics; this is an essential capability for handling languages such as Ada, where the semantics is not yet fixed and is parameterized on various data types. We outline in the paper why it is significant and how it is possible to define within a coherent schema various semantics ranging from an initial algebraic semantics to a variety of semantics depending on the observations we want to make of a system.

Combining the chosen semantics of the intermediate language with the first step (formally this corresponds to the composition of two homomorphisms) results in an overall denotational semantics for the source language.

The purpose of this paper is twofold: to provide a readable introduction to the basic ideas of a compositional and denotational treatment of concurrent languages and to outline the overall technical structure of the SMoLCS approach using a direct semantics style. Hence the paper has two parts, the first more introductory and the second more technical.

For illustrating the technical treatment we use a toy language CL as a running example. We emphasize that CL has been chosen following the usual paradigm for illustrating a methodology: It contains all and only the essential constructs for explaining most of the relevant technical features. Hence, CL should not be considered more than a paradigm.

There is unfortunately one important capability of our method which cannot be exemplified by CL, just because of its simplicity. Our approach is highly modular and parameterized, so that it can handle languages and systems of any complexity, with modules and even with parts which are not completely specified. For these aspects we can only illustrate the technical approach, showing its application to CL, without pretending that the example shows their importance. However, we are confident that the reader will understand from our presentation the generality of the approach. Nonetheless, if there is any doubt, it should be enough to recall that the same methodology has been used for the full formal definition of Ada [8], which is by far the largest formal specification of a language ever written and the first to be proved adequate to handle such a complex language as Ada.

The plan of the paper is the following. In the second section we recall, to help the reader, the main features of denotational semantics for the sequential case. In the third we discuss informally the problems posed by the presence of concurrent constructs, we present the main concepts for modeling processes, and we introduce, rather informally, the basic ideas of our approach. In the fourth section we present both an informal introduction and a completely formal specification of the intermediate language using the techniques of partial algebraic specifications. Finally, in the fifth section the denotational semantics of the example language is given. In an appendix we collect the basic terminology for understanding the algebraic aspects. The importance attached to the intermediate language is justified by the fact that it differs only in size from the one used for the formal definition of full Ada, and hence it can be taken as a basis for the definition of a wide range of concurrent systems and languages. This is a point emphasized in a further work [9], where we have proposed a language (with its semantics) based on an extension of the intermediate language proposed here.

2. Denotational semantics for sequential languages

- *The example language CL*

In this subsection we introduce the simple concurrent

programming language CL, which is used as an example for describing our methodology.

The abstract syntax of CL is given in the usual BNF style:

```
 1 PROG ::=       program BLOCK {|| BLOCK}
 2 BLOCK ::=      DECS begin STAT HANDLERS end
 3 DECS ::=       Λ | var VID DECS
 4 HANDLERS ::=   Λ | when EID do STAT HANDLERS
 5 EXP ::=        CONST | VID | BOP(EXP,EXP)
 6 STAT ::=       VID := EXP | STAT; STAT | BLOCK |
 7                if EXP then STAT else STAT fi |
 8                while EXP do STAT od |
 9                raise EID |
10                send(CHID,EXP) | rec(CHID,VID) |
11                create task BLOCK | STAT or STAT
```

Identifiers of variables (VID), exceptions (EID), channels (CHID), symbols of constants (CONST), and binary operators (BOP) are nonterminal symbols which are not further specified.

CL is a block-structured language (2) and blocks can be nested (6); moreover, it has the usual sequential statements (6, 7, 8). To simplify the paper we have not completely defined CL expressions; note, however, that they include Boolean expressions.

CL has an (Ada-like) exception mechanism (2, 4, 9); every block has a handlers part, and when an exception *ei* is raised in a block by a raise statement, the execution of the block is abandoned and, if for some statement *st*, when *ei* do *st* appears in the handlers part of that block, then the statement *st* is executed; otherwise, the exception *ei* is propagated outside the block.

A CL program consists of a set of tasks in parallel, and tasks are just blocks (1); moreover, in CL there is the possibility of creating new tasks by means of the statement create task (11); all tasks are executed in parallel, and there is no constraint on the duration of the execution of their statements.

The variables declared in a block (2) are shared by the block itself and by the tasks created within the block. Clearly, two tasks cannot update the same variable simultaneously.

Tasks can also exchange messages through channels, by a handshaking-like mechanism; i.e., a task can execute a statement rec(*ci*,*x*) only if some other task can execute simultaneously a statement send(*ci*,*e*) and vice versa; as a result of the execution of the two statements, the value of the expression *e* will be received by the first task and assigned to the variable *x*.

Moreover, there is also a statement for nondeterministic choice; a task executing st_1 or st_2 could choose nondeterministically to execute either the statement st_1 (if possible) or the statement st_2 (if possible); the choice in CL corresponds neither to global nondeterminism nor to local nondeterminism; it depends on the form of the statements st_1 and st_2 [e.g., it is local in the case of create task bl_1 or create task bl_2; it is global in the case of rec(ci_1,x) or rec(ci_2,x)].

Note that in CL there are no mechanisms for declaring exceptions and channel identifiers; only variable identifiers must be declared before being used.

If we drop the concurrent structure from CL (the statements of lines 10 and 11 and the construct ||), we obtain a classical sequential language, called the *purely sequential subset* of CL, with the usual well-known semantics. But note that, in general, CL statements such as assignment, conditional, and while are not sequential statements because now, due to the presence of shared variables, their executions require concurrent interactions with other tasks. For example, given two shared variables x and y, the execution of the statement $x := 0$ could be delayed forever because other tasks continue updating x forever (CL does not require fairness in getting access to the shared variables); moreover, the value assigned to x by the execution of $x := y$ depends on the moment when the statement is executed, because in the meantime other tasks could have updated y.

• *Principles of denotational semantics*

Consider a sequential language consisting of declarations, expressions, and statements as the purely sequential subset of CL. Then the *denotational* semantics consists in associating a *denoted* value (the meaning) with each declaration, expression, and statement. This is done by defining three functions, called *semantic functions*:

D: DECS $\rightarrow$ *DEC-VAL*

E: EXP $\rightarrow$ *EXP-VAL*

S: STAT $\rightarrow$ *STAT-VAL*

where DECS, EXP, and STAT are the sets of syntactic objects representing declarations, expressions, and statements, called *syntactic domains*, and *DEC-VAL*, *EXP-VAL*, and *STAT-VAL* are the corresponding sets of denoted values, called *semantic domains*.

Now, in order to define, in accordance with our intended informal meaning, the semantic domains, we need some auxiliary structures, called *auxiliary domains*. For standard denotational semantics these are the domains *ENV* and *STORE*. *ENV* is the domain of environments, which are (partial) functions from identifiers to the values denoted by such identifiers (*DEN*):

$ENV = (\mathrm{ID} \rightarrow DEN)$.

STORE is the domain of stores (or memories), which are functions from locations (*LOC*) to storable values (*VAL*):

$STORE = (LOC \rightarrow VAL)$.

The value denoted by a CL variable identifier is a location in the store.

Then it is natural to define, at least for a simple language (e.g., the purely sequential subset of CL without the exception mechanism),

$$DEC\text{-}VAL = (ENV \rightarrow ENV)$$

$$EXP\text{-}VAL = ((ENV \times STORE) \rightarrow VAL)$$

$$STAT\text{-}VAL = ((ENV \times STORE) \rightarrow STORE).$$

Indeed, for example, a statement, given an environment and a store, may produce a change in the store, hence another store (consider, e.g., an assignment).

It is convenient and has become common to write, for example, $(ENV \rightarrow (STORE \rightarrow STORE))$ instead of $((ENV \times STORE) \rightarrow STORE)$; this use corresponds to the so-called "currying" technique, which reduces functions of many arguments to a nested chain of functions of one argument. In the example above, the value of a statement will be a function which, given an environment, produces another function, which, given a store, produces a store.

Thus, for example, the value of an assignment can be expressed by the clause

i) $\mathbf{S}[x := e]\rho\sigma = \sigma[\mathbf{E}[e]\rho\sigma/\rho(x)]$

which says that the value of the statement $x := e$, given an environment ρ and a store σ, is a new store obtained from σ by updating the location $\rho(x)$ with the value of e, which in turn is obtained as the result of the function **E** applied to e, ρ, and σ, i.e., the value of e with respect to the environment ρ and the store σ.

Here we adopt the usual notation: Square parentheses are used, instead of round ones, around elements of the syntactic domains, and we do not put parentheses around curried arguments (for example, $\mathbf{S}[x := e]\rho\sigma$ stands for $((\mathbf{S}(x := e))(\rho))(\sigma)$) and, given a function f, $f[a/b]$ stands for λx. **if** $x=b$ **then** a **else** $f(x)$ (here and in the following we use the well-known λ-notation for expressing functions: e.g., a function $f: x \rightarrow x^2$ is indicated by $\lambda x.x^2$).

Clearly, clause i) can be written equivalently as

ii) $\mathbf{S}[x := e] = \lambda\rho,\sigma.\sigma[\mathbf{E}[e]\rho\sigma/\rho(x)]$

which gives explicitly the value of the statement $x := e$.

Moreover, the semantic functions are defined by compositionality. Informally, compositionality means that every well-formed syntactic construct is given a meaning depending only on the meaning of its subconstructs and on the meaning of the syntactic operator building that construct out of its subconstructs. We make precise the idea using again the assignment statement.

An assignment statement $x := e$ is built using the assignment operator := and the two subconstructs x, which is a variable identifier, and e, which is an expression. Then, giving the semantics of $x := e$ by compositionality amounts to saying that the meaning of $x := e$ is the result of the meaning of the operator :=, say $\mathbf{M}(:=)$, applied to the meaning of its arguments $\mathbf{M}(x)$ and $\mathbf{M}(e)$; formally we should write

iii) $\mathbf{S}[x := e] = \mathbf{M}(:=)(\mathbf{M}(x),\mathbf{M}(e))$.

Now $\mathbf{M}(x)$ is a function which, given an environment ρ, produces the value of the identifier $\rho(x)$ (i.e., a location), and $\mathbf{M}(e)$ is just a function which, given an environment ρ and a store σ, gives as result $\mathbf{E}[e]\rho\sigma$. Formally we can write

iv) $\mathbf{M}(x) = \lambda\rho.\rho(x)$, $\mathbf{M}(e) = \mathbf{E}[e] = \lambda\rho,\sigma.\mathbf{E}[e]\rho\sigma$.

Hence, clause i) is equivalent to saying that $\mathbf{M}(:=)$ is a function which, given an element $\mathbf{M}(x) \in (ENV \rightarrow LOC)$ and an element $\mathbf{M}(e) = \mathbf{E}[e] \in ((ENV \times STORE) \rightarrow VAL)$, gives as result a function, specifically $\mathbf{S}[x := e]$, which, given ρ and σ, has result $\sigma[\mathbf{E}[e]\rho\sigma/\rho(x)] = \sigma[\mathbf{M}(e)\rho\sigma/\mathbf{M}(x)\rho]$. In other words, assuming iv), the clauses ii) and iii) are two equivalent ways of defining the semantics of assignment by compositionality.

A last but most important point here is to understand that the overall set of semantic clauses, defining the functions **E**, **D**, and **S**, is given by induction on the syntactic structure of the constructs, or equivalently, that we have one clause for each syntactic construct, where the meaning of the subconstructs is given in turn using the same semantic functions **E**, **D**, and **S** (together with other known functions, of course).

As another example, consider the clause for the concatenation of two statements:

$$\mathbf{S}[st_1;\ st_2]\rho\sigma = \mathbf{S}[st_2]\rho(\mathbf{S}[st_1]\rho\sigma).$$

That is equivalent to the two following clauses:

$$\mathbf{M}(st_1;\ st_2) = \mathbf{M}(;)(\mathbf{M}(st_1),\mathbf{M}(st_2))$$

$$\mathbf{M}(;) = \lambda f,g.(\lambda\rho,\sigma.g(\rho, f(\rho,\sigma))).$$

Summarizing, at a rather informal level denotational semantics consists in defining a denoted value for each well-formed construct in a compositional way. At a bit more technical level, a denotational semantics is given by defining, inductively on the syntactic structure of constructs, a set of semantic functions, one for each type or sort of construct, after having defined the appropriate semantic domains.

Until now, we have seen that in simple cases it is possible to give a nice denotational semantics following a direct style, i.e., where every construct has a straightforward meaning which is associated directly with the construct. Problems arise when defining constructs that change the normal flow of execution, for example, the exception mechanisms of CL, subprogram returns, goto's, and so on. To handle them two methods have been developed:

- Use of continuations by the Oxford school (Strachey and Wadsworth; see, e.g., [4, 5]).

- Use of an exit-trap mechanism by the VDM school (Bjørner and Jones [6]).

In the Oxford style the semantic function for statements takes another argument, the so-called continuation, which represents the meaning of what follows a statement and is a store transformation; i.e., we have

$$\mathbf{S}: \text{STAT} \rightarrow (ENV \rightarrow (CONT \rightarrow CONT)),$$

where $CONT = (STORE \rightarrow STORE)$. Thus, the meaning of a statement is given indirectly depending on the context of what follows it.

The clause for statement concatenation will have the form

$$\mathbf{S}[st_1;\ st_2]\rho\theta = \mathbf{S}[st_1]\rho(\mathbf{S}[st_2]\rho\theta).$$

The VDM school tries instead to keep a direct style, changing the meaning of a statement, which clearly cannot be a store transformation anymore. The basic idea is to have

$$\mathbf{S}: \text{STAT} \rightarrow (ENV \rightarrow STAT\text{-}VAL)$$

where $STAT\text{-}VAL = (STORE \rightarrow (STORE \times FOLLOW))$ and $FOLLOW = \text{EID} \cup \{\text{next}\}$. (EID is the set of the exception identifiers and next is a special identifier not belonging to EID.)

If $\mathbf{S}[st]\rho\sigma = \langle \sigma', f \rangle$, then f indicates the next point in the execution flow after the execution of st; specifically, we have $f =$ next when the execution follows the normal flow and $f = ei$ when the exception ei is raised.

For readability reasons the VDM school has defined some operators on $STAT\text{-}VAL$:

- **exit** ei corresponding to $\lambda\sigma.\langle \sigma, ei \rangle$;
- $st\text{-}val_1; st\text{-}val_2$ corresponding to
  $\lambda\sigma.$**let** $\langle \sigma', f \rangle = st\text{-}val_1(\sigma)$ **in**
  if $f =$ next **then** $st\text{-}val_2(\sigma')$ **else** $\langle \sigma', f \rangle$;
- **trap** $emap$ **in** $st\text{-}val$, where $emap$ is a map from exception identifiers into statement values, corresponding to
  $\lambda\sigma.$**let** $\langle \sigma', f \rangle = st\text{-}val(\sigma)$ **in**
  if $f \in \text{Dom}(emap)$ **then** $emap(f)\sigma'$ **else** $\langle \sigma', f \rangle$.

Now statement concatenation and exceptions can be handled as follows:

$$\mathbf{S}[st_1;\ st_2]\rho = \mathbf{S}[st_1]\rho; \mathbf{S}[st_2]\rho$$

$$\mathbf{S}[\underline{\text{raise}}\ ei]\rho = \textbf{exit}\ ei$$

$$\mathbf{S}[\underline{\text{begin}}\ st\ \underline{\text{when}}\ ei_1\ \underline{\text{do}}\ st_1 \cdots \underline{\text{when}}\ ei_n\ \underline{\text{do}}\ st_n\ \underline{\text{end}}]\rho =$$

$$\textbf{trap}\ [ei_1 \rightarrow \mathbf{S}[st_1]\rho, \cdots, ei_n \rightarrow \mathbf{S}[st_n]\rho]\ \textbf{in}\ \mathbf{S}[st]\rho.$$

Similarly, we can handle the semantics of expressions and declarations, whenever they have side effects. In order to show the basic idea, let us briefly illustrate the case of expressions.

In the Oxford style the semantics of expressions is handled by using a class of continuations, specifically the expression continuations, which represent the meaning of what follows an expression; formally they are functions which, given a value (the value of that expression), produce some store transformation (i.e., a statement continuation)

$$ECONT = (VAL \rightarrow CONT)$$

Consequently, the semantic function for expressions now has functionality

$$\mathbf{E}: \text{EXP} \rightarrow (ENV \rightarrow (ECONT \rightarrow CONT))$$

and the clause for the assignment has form

$$\mathbf{S}[x := e]\rho\theta = \mathbf{E}[e]\rho(\lambda v, \sigma.\theta(\sigma[v/\rho(x)]))$$

where $\lambda v, \sigma.\theta(\sigma[v/\rho(x)])$ is an expression continuation.

Following the direct semantics style, the meaning of an expression is an element of $(STORE \rightarrow (STORE \times VAL))$, and for readability purposes, some special operators are defined for handling these expression meanings; let us now recall some of them.

Let $f \in (STORE \rightarrow (STORE \times VAL))$,

$g \in (VAL \rightarrow STAT\text{-}VAL)$:

- **def** f **in** g corresponds to

  $\lambda\sigma.$**let** $\langle \sigma', v \rangle = f(\sigma)$ **in** $(g(v))(\sigma') \in STAT\text{-}VAL$;

- **return** v, where $v \in VAL$, corresponds to $\lambda\sigma.\langle \sigma, v \rangle$.

In this framework we have that the semantic function for expressions has functionality

$$\mathbf{E}: \text{EXP} \rightarrow (ENV \rightarrow (STORE \rightarrow (STORE \times VAL)))$$

and the clause for the assignment has form

$$\mathbf{S}[x := e]\rho = \textbf{def}\ \mathbf{E}[e]\rho\ \textbf{in}\ \lambda v, \sigma.\langle \sigma[v/x], \text{next} \rangle.$$

The semantic style using continuation is illustrated in [4, 5] and a semantics of a language in the SMoLCS approach using continuation is presented in [7]. In the following we consider only the direct semantics style with VDM-like combinators, since that is the style used in the formal definition of Ada.

3. Handling concurrency in a denotational framework

- *Problems*

Several problems arise when we try to extend the denotational approach to concurrent languages. Indeed, the semantics of a language construct can no longer be a function representing a state transformation; consider, for example, the CL statement $x{:=}1$ $\underline{\text{or}}$ $x{:=}2$; what is its semantics in the environment ρ? Perhaps it could be the nondeterministic function $\lambda\sigma.\{\sigma[1/\rho(x)], \sigma[2/\rho(x)]\}$. However,

the semantics of (x:=1 or x:=2) ‖ y:=3 must be obtained in a compositional way from the semantics of the component tasks; i.e., we want to have

$$\mathbf{M}((x{:=}1 \text{ or } x{:=}2) \parallel y{:=}3) = \mathbf{M}(\parallel)(\mathbf{M}(x{:=}1 \text{ or } x{:=}2), \mathbf{M}(y{:=}3)).$$

In the simple case above, a solution can be found: $\mathbf{M}(\parallel)$ is an operator which composes two functions in the two possible orders, but there are no solutions for the case of $tk_1 \parallel tk_2$, where $tk_1 = x{:=}y; st_1$ and $tk_2 = y{:=}1; y{:=}y{+}1; st_2$. The value of y during the execution of tk_1 depends on the part of tk_2 which has been already executed. That means that the execution of tk_1 can have several effects on the store depending on the (unknown) value of y at that time; hence the semantics of tk_1 must express this kind of nondeterminism.

In general it can be easily understood from the kind of arguments above that we have to model the fact that the execution of a statement depends on the interaction with an outside context, i.e., a context which contains no more "private information" than the store for the sequential case, but is influenced by the execution of other parts of the program. It follows, therefore, that we have to model in some sense which are the "potential" executions of a statement (we will speak of capabilities of performing actions), which will or will not become actual executions depending on the context.

Another illustrative example is the following. Assume that st_1 = rec(ci,x) and st_2 = send(ci,e). It is clear that in this case the execution of tk_1 is influenced by the execution of tk_2 (and conversely) in a stronger way. We can say that tk_1 has the capability of performing the action corresponding to the execution of the rec statement and that this capability becomes effective only if tk_2 performs simultaneously the action corresponding to the execution of the send statement. But what is a precise formalization of "performing simultaneously"?

It should be clear that it is rather complicated (and indeed it has been shown to be impossible) to model all these aspects of concurrency by using just functions. In the following subsections we present an approach which can resolve elegantly these problems and allows us to maintain a compositional style in modeling concurrent languages. We need a formal model which can play for concurrency the role of functions, so that we can assign a denoted value to each well-formed fragment of a concurrent language.

In our approach, inspired by CCS [10] and SOS [11], this model is a labeled transition system, which will be in the following the formal counterpart of an informal or purely syntactic notion of concurrent process.

- *Modeling processes as labeled trees*

Labeled transition systems and labeled trees A *labeled transition system* is a set of triples (s,f,s'); a triple is also written $s \xrightarrow{f} s'$ and means that the system has the capability of passing from the state s to the state s' under an interaction with the external environment represented by the label (or flag) f. In the simplest case, when the transition is purely internal to the system and there is no relationship with the environment, the label can be dropped or better represented by a special label, which is usually written TAU as in CCS.

For example, the capabilities associated with a rec(ci,x) statement could be represented by a set of labeled transitions of the form $s \xrightarrow{\text{REC}(ci,v)} s'$ (one for each v in the set of values which can be received), which means that the process representing rec(ci,x) can pass from the state s [corresponding to the situation immediately before the execution of rec(ci,x)] to a state s' (a state in which, for example, it is recorded that the value v has been received and must be assigned to the shared variable x) performing an action of receiving v from the outside on the channel ci; obviously, in the state s there is one capability for each value v, and that expresses exactly the fact that v is received from the outside.

The capability associated with a send(ci,v) statement could be represented by the labeled transition $s \xrightarrow{\text{SEND}(ci,v)} s'$, which means that the process representing send(ci,v) can pass from the state s [corresponding to the situation immediately before the execution of send(ci,v)] to the state s' performing the action of sending the value v outside on the channel ci.

Given a transition system, with each state s is associated a *labeled tree*, in which we do not care about permutation of branches, and two equal subtrees with the same root are considered once (in the following, when speaking of labeled trees, we will always consider them up to this equivalence).

On labeled trees branching represents nondeterminism. As a classical example of a nondeterministic process, consider a memory register which can either be written or read; i.e., it can either receive a value from outside or send out its content (see [10]):

$$\text{Reg}(v) = (\mathbf{choose}_{u \in \text{VAL}}\ \text{IN}(u) \ \Delta\ \text{Reg}(u)) + \text{OUT}(v) \ \Delta\ \text{Reg}(v)$$

Here we have introduced a simple syntax for expressing processes: + represents nondeterministic choice, $\mathbf{choose}_{u \in \text{VAL}}$ means a choice between the elements in VAL, Δ means "followed by," and IN($\cdots$), OUT($\cdots$) correspond to the two capabilities of the register of being, respectively, written or read. The corresponding tree is in **Figure 1.**

Note that the definition of Reg(v) is recursive, and hence a process parameterized on a value such as IN(u) Δ Reg(u) is represented by an infinite branching tree, with one branch for each value.

Composing labeled trees We need also to model groups of interacting processes; in these cases we use a particular class of transition system: concurrent systems.

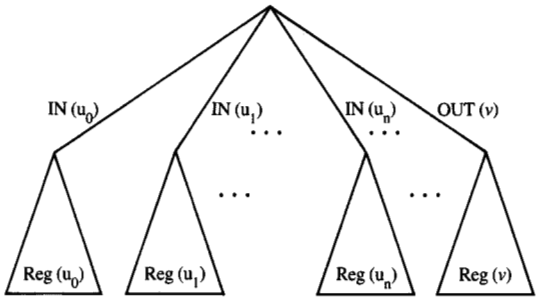

Figure 1

Labeled tree associated with Reg (v).

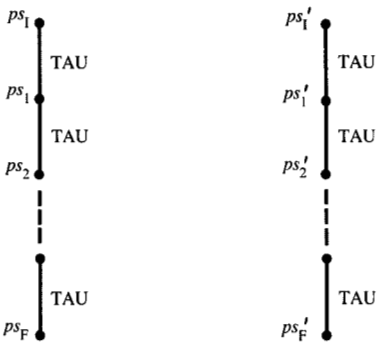

Figure 2

Labeled trees associated with sequential processes.

A *concurrent system* is a labeled transition system built from some component subsystems; each subsystem is in turn modeled as a labeled transition system.

A state of a concurrent system is modeled as a set of states corresponding to the subsystems plus some global information; the transitions are inferred from the transitions of the component subsystems.

For example, the parallel composition of two processes p_1 and p_2 (which can interact by means of a shared store and exchanging messages through channels) could be represented by the following state of concurrent system $cs = \langle\{ps_1,ps_2\},sh\rangle$, where ps_1 and ps_2 represent the initial states of the processes p_1,p_2 and the global information part sh represents the state of the shared store.

Let us assume that $ps_1 \xrightarrow{\text{SEND}(ci,v)} ps_1'$ and $ps_2 \xrightarrow{\text{REC}(ci,v)} ps_2'$ are two transitions of p_1 and p_2. Starting from these transitions of the subcomponents, the following transition of cs corresponds to the synchronized exchange of a message between p_1 and p_2:

$$\langle\{ps_1,ps_2\},sh\rangle \xrightarrow{\text{COMM}(ci)} \langle\{ps_1',ps_2'\},sh\rangle.$$

Note that the state of the shared store has not been changed by the message exchange.

To give an example of a transition of the concurrent system where the global information is changed, let us consider the case when the process p_1 has also the capability $ps_1 \xrightarrow{\text{WRITE}(x,v)} ps_1''$ (due, for example, to the execution of an assignment to the shared variable x):

$$\langle\{ps_1,ps_2\},sh\rangle \xrightarrow{\text{WRITE}(x,v)} \langle\{ps_1'',ps_2\},sh[v/x]\rangle$$

(here the process p_2 has not taken part in the action).

Nondeterminism can also come from parallelism, for example, when a process can perform some actions with at least two other processes. Consider a process which has the ability of receiving a message (e.g., p_2 above) and there are in parallel two other processes able to send it (e.g., p_1 above and p_3 having the capability $ps_3 \xrightarrow{\text{SEND}(ci,v)} ps_3'$); clearly, there are at least two exclusive possible transitions of the system in that situation. Specifically,

$$\langle\{ps_1,ps_2,ps_3\},sh\rangle \xrightarrow{\text{COMM}(ci)} \langle\{ps_1',ps_2',ps_3\},sh\rangle$$

and

$$\langle\{ps_1,ps_2,ps_3\},sh\rangle \xrightarrow{\text{COMM}(ci)} \langle\{ps_1',ps_2,ps_3'\},sh\rangle.$$

Labeled trees and their semantics Representing processes or groups of processes in parallel as labeled trees is already a step toward giving a semantics to concurrency. However, the difficult problem in semantics is abstraction; i.e., to give a semantics sufficiently abstract with respect to linguistic details. In this respect, while labeled trees are much more abstract than pieces of code, it is nevertheless clear that in many cases there are different trees with the same semantics. One of the simplest cases is when we consider processes corresponding to purely sequential commands; their models as labeled trees are unary trees with all the arcs labeled by the symbol of internal action TAU, as in **Figure 2.**

Then, if we are interested in an input-output semantics, we would say that the two trees (here sequences) are equivalent iff ps_I and ps_I' are input-equivalent and ps_F and ps_F' are output-equivalent, irrespective of differences in other aspects, such as the intermediate states.

From this simple example we can understand that a semantics is then given by an appropriate equivalence on labeled trees. The case we have just discussed informally shows that the functional semantics for sequential languages can be obtained as a special case of semantics for concurrent languages, just by considering two trees (now just unary trees like those seen in Figure 2) semantically equal whenever they are input-output equivalent.

Thus, it is easily understood that other interesting equivalences on trees can arise, depending in general on what we want to observe of a system. For example, two well-known equivalences are strong equivalence (see [10]) and stream (trace) equivalence. For the first, we consider two trees equivalent iff they are the same tree by forgetting the states (always modulo the equivalence defined in the beginning). For the second, two trees are equivalent iff the two corresponding sets of label sequences starting from the two roots are the same.

- *General structure of SMoLCS*

The principles and basic ideas sketched above for handling concurrency in a compositional way are formally expressed within the SMoLCS methodology, whose overall structure is now outlined.

SMoLCS is an integrated methodology for the specification of concurrent systems and languages developed mainly at the University of Genoa (Department of Mathematics) by the authors (see [7, 12, 13]), and relying for the algebraic setting on cooperation with M. Wirsing (Passau–Fakultät für Informatik) (see [14, 15]). The typical fields of application of SMoLCS are large systems, multilevel architectures built from systems with different granularity, and complex concurrent languages with modules and interference between sequential and concurrent features.

SMoLCS consists of four parts: *specification of concurrent systems, semantics of concurrent languages, metalanguage,* and *tools*. Its main features are the following.

Specification of concurrent systems A concurrent system is a labeled transition system built from some component subsystems; each subsystem is in turn modeled as a labeled transition system.

A state of a concurrent system is modeled as a set of states corresponding to the subsystems plus some global information; the transitions are inferred from the transitions of the subsystems in three steps—synchronization, parallelism, and monitoring:

- *Synchronization* defines the transitions representing synchronized actions of sets of subsystems and their effects on global information.
- *Parallelism* defines the transitions representing admissible parallel executions of sets of synchronized actions and the compound transformations of the global information (mutual exclusion problems, for example, are handled here).
- *Monitoring* defines the transitions of the overall system respecting some abstract global constraints (such as interleaving, free parallelism, priorities, etc.).

Now we can explain what SMoLCS means: SMoLCS stands for Structured Monitored Linear Concurrent Systems; linear means that the actions of a SMoLCS concurrent system consist in performing in parallel groups of actions of the component subsystems.

This SMoLCS schema is expressed in an algebraic way so that the transition system corresponding to the component subsystems, the global information, and the whole concurrent system are abstract data types, and also the three steps are specified by giving appropriate abstract data types. Moreover, the overall SMoLCS schema is formalized as a parameterized abstract data type, where the parameters define the systems to be composed and their interactions. Thus every SMoLCS definition of a specific system will be an instantiation of that parameterized specification.

Clearly, since a concurrent system is the specification of a labeled transition system, it can be taken as a basic transition system for another SMoLCS specification; that means that the SMoLCS construction can be iterated. Moreover, it is possible to give a SMoLCS schema corresponding to an inductive definition.

Since a SMoLCS specification is the specification of an abstract data type, we can consider, as usual, some well-known classes of semantics, which in the case of concurrent systems have a special meaning. For example, we can consider an initial algebra semantics, from which an operational semantics can easily be derived. Moreover, the SMoLCS approach supports the definition of an observational semantics via a parameterized abstract data type specification, where the parameters correspond to a formalization of the observations. Every instantiation of such a schema admits a terminal model, the *concurrent algebra*, in which two states of the concurrent system are equivalent if and only if they satisfy the same observations; moreover, every subpart of the state gets an observational semantics by closure with respect to state contexts (see [15] for foundations). The above schema permits us to formalize observationally various semantics, such as input/output, strong equivalence, classes of bisimulation equivalences (see [16]), and stream semantics.

Semantics of concurrent languages The SMoLCS methodology is based on a two-step approach.

Essentially the first step connects the abstract syntax of the source language to an underlying model for concurrency, formalized in a language suitable for describing processes and their mutual interactions in a concurrent system. This is done by a set of *denotational clauses*, where in a typical

denotational style a term in an intermediate language is associated with each well-formed construct of the source language. For example, a function such as

$\mathbf{P}$: PROG $\rightarrow$ *STATE*

will associate with a program a term representing a state of a concurrent system corresponding to the executions of programs.

A function such as

$\mathbf{S}$: STAT $\rightarrow$ (*ENV* $\rightarrow$ *PROC*)

will associate with a statement, given an environment, a term representing a process, i.e., a state of a transition system corresponding to the activity of the process to which the statement belongs. The states of the system corresponding to the processes (the subcomponents of the concurrent system mentioned before) are given as a part of an algebraic specification which formalizes a labeled transition system. Declarations and expressions are handled analogously.

Altogether the denotational clauses can be seen as defining, inductively on the structure of the abstract syntax, a syntax-directed translation into an intermediate language for representing processes and concurrent systems.

The semantics of the intermediate language is given, following the SMoLCS technique for the specification of concurrent systems, by the algebraic specification of a *concurrent algebra* (the second step), representing a concurrent system modeling program executions. As we have seen before, the specification of the concurrent algebra consists essentially in

- The specification of an abstract data type formalizing a concurrent system as a labeled transition system.
- An observational semantics associated with that abstract data type.

Thus the terms of the intermediate language, obtained by the denotational clauses in the first step, can be interpreted in the concurrent algebra. In this way the denotational clauses define a homomorphism from the algebra of the abstract syntax into a semantic algebra, some carriers of which are the carriers of the corresponding sorts in the concurrent algebra.

Metalanguage The metalanguage consists of two kernels, one applicative and one algebraic, with a simple semantics consisting of a simple connection between the semantics of the two kernels, which are given following classical methods. Actually, with SMoLCS is associated a metalanguage schema, which has to be filled depending on the SMoLCS parameters. An example of such metalanguages is the metalanguage M used in the formal definition of Ada [8].

Tools So far a specific rapid prototyping tool has been realized for SMoLCS ([17]) which is an extension of the RAP system [18], specially tailored to the structure of SMoLCS. It consists of a concurrent symbolic interpreter, which can derive transitions for a specified concurrent system, and of an interpreter for denotational clauses.

4. The intermediate language

As we have outlined, in our approach the definition of the semantics of a language consists of a set of denotational clauses, which can be seen as a translation into an intermediate language.

In this section we give both an informal description and a formal definition of the intermediate language called ***SYST***, which is used in the following section for defining the semantics of concurrent languages following a direct style. The emphasis we put on ***SYST*** is justified by the fact that it can be used for defining the semantics of languages of any complexity, just by specifying some parameters in more detail. For example, in essence it is the intermediate language used for the definition of Ada.

- *Introducing the intermediate language*

We first introduce the syntax of the intermediate language, with some informal comments on its semantics, to prepare the reader to understand the full formal specification given later.

The concepts needed for understanding this section have been given in Section 3.

The intermediate language is designed for representing concurrent systems; thus ***SYST*** describes the states of a concurrent transition system indicated by SYST. A state of a concurrent system is described by giving the states of its subcomponent processes and its global information part; thus, first we give the language constructs for expressing the global information parts and the process states and then the construct for composing them into states of the concurrent systems. The states of the subcomponent processes of SYST are called *behaviors* and are represented by elements of type behavior.

The syntax of the operators is as follows:

Op: $\text{elem}_1 \times \cdots \times \text{elem}_n \rightarrow \text{elem}$

which says that the operator Op, given the objects $o_1 \cdots o_n$ of type $\text{elem}_1 \cdots \text{elem}_n$, respectively, returns an object of type elem, represented by the expression $\text{Op}(o_1, \cdots, o_n)$.

Global information

In the case of SYST the global information corresponds to a store shared between the subcomponent processes; its states (elements of type store) are represented as finite mappings of locations into values. Here we do not give a complete definition of the values (elements of type val); note only that they include Boolean values.

- True, False: → val
  are the operators for expressing truth values.

Also, the locations of the shared store (elements of type loc) are not further specified.

The operators for expressing store states are as follows:

- []: → store
  This represents the empty store, i.e., the map with empty domain.
- □(□): store × loc → val
  Given a store state *sh* and a location *l*, *sh*(*l*) represents the content of the location *l* in *sh*, i.e., the value associated with *l* by the map *sh*.
- □[□/□]: store × val × loc → store
  Given a store state *sh*, a value *v*, and a location *l*, *sh*[*v*/*l*] represents the store state in which the content of *l* is *v* and the content of a location $l' \neq l$ is the same as in *sh*.
- Dom: store → set(loc)
  Given a store state *sh*, Dom(*sh*) represents the set of the locations used in *sh*, i.e., the domain of the map *sh*.

Behaviors

First we give the operators for expressing the atomic actions of the component processes of SYST (elements of type act).

Processes can perform internal actions, i.e., actions which do not require interactions either with other processes or with the shared store; these actions are represented by

A1) TAU: → act.

Processes can allocate new cells of the shared store:

A2) ALLOC: loc → act.

The action ALLOC(*l*) can be performed only if the location *l* is still unused in the actual state of the store.

Processes can write and read the contents of the cells of the shared store:

A3) WRITE, READ: loc × val → act.

Clearly, a cell can be written or read only if it has been previously allocated, and an action READ(*l*,*v*) can be performed only if the actual content of the cell individuated by the location *l* is equal to *v*.

Processes can exchange messages (values) through channels (individuated by elements of type chid):

A4) SEND, REC: chid × val → act.

This kind of communication is handshaking; thus, a process can perform an action SEND(*ci*,*v*) only together with another process which performs REC(*ci*,*v*) and vice versa.

A process can create some new process:

A5) CREAT: behavior → act

After a process *p* has performed an action CREAT(*bh*), the process individuated by *bh* will perform its actions in parallel with *p* and with the other processes which were in parallel with *p*.

Then we list the operators for expressing behaviors:

B0) **skip:** → behavior

skip represents the null process, i.e., the process unable to perform any action.

B1) □ Δ □: act × behavior → behavior

a Δ *bh* represents the process which performs the action *a* and then behaves as specified by *bh*.

The Δ combinator is the basic tool for expressing the activity of a process as a sequence of atomic actions; it corresponds to the CCS dot.

In what follows, given a type arg, funct(arg,behavior) indicates the type of the functions from arg into behaviors, and we have the following operators:

F1) λ□.□: arg-var × behavior → funct(arg,behavior) (abstraction)

The elements of type arg-var represent in some way the "variables of type arg." There is also an operator which embeds these "variables" into the elements of type arg

F2) Arg-Var: arg-var → arg

and various operators for expressing the elements of type arg-var.

For every element of type arg-var *v*, Arg-Var(*v*) is simply written *v*; moreover, every string of lowercase letters corresponds to an element of type arg-var.

F3) □(□): funct(arg,behavior) × arg → behavior (application)

B2) For arg = val, loc, bool (bool is the type of the Boolean values)

$\mathbf{choose}_{arg}$ □: funct(arg,behavior) → behavior

$\mathbf{choose}_{arg}$ *bhfunct* represents the process which can nondeterministically behave as specified by *bhfunct*(*a*) for every element of type arg *a*.

The importance and relevance of these combinators for representing subcomponent processes of concurrent systems should be clear (see, e.g., [10, 16]). Following Milner's notations (see, e.g., [16]) we would write these combinators as $\sum_{a \in ARG} bh(a)$, where ARG is a set and *bh*(*a*) is a behavior expression parameterized on *a* (i.e., an expression of type behavior with a free variable *a* of type ARG). Here, we have chosen to consider a combinator $\mathbf{choose}_{arg}$ applied to functions from elements of type arg into behaviors; thus, the parameterized dependence of *bh*(*a*) on *a* is formally expressed by means of an element of type funct(arg,behavior); hence $\sum_{a \in ARG} bh(a)$ will be written $\mathbf{choose}_{arg} \lambda a.bh(a)$. Our nondeterministic choice is neither local nor global; that depends on the nature of the alternatives of the choice. For example,

- If for every element a of type arg, the first-step actions of $bh(a)$ correspond to interactions with other processes or the global information, then we have global nondeterminism (e.g., for bh = **choose**$_{val}\lambda v$.REC(ci,v) Δ bh_1, if a process can send the value v_o along the channel ci, then bh will choose the alternative REC(ci,v_o) Δ bh_1).
- If for every element a of type arg, the first-step actions of $bh(a)$ are all internal actions, then we have local nondeterminism (e.g., if bh = **choose**$_{val}\lambda$v.TAU Δ SEND(ci,v) Δ bh_1, then bh can choose one of the alternatives independently from the external context).

B3) **if** □ **then** □ **else** □:
val × behavior × behavior → behavior

The usual conditional operator: **if** bv **then** bh_1 **else** bh_2 is equal to bh_1 when bv is equal to True and it is equal to bh_2 when bv is equal to False.

In what follows we write $bh_1 + bh_2$ instead of **choose**$_{bool}$ λ b. **if** b **then** bh_1 **else** bh_2 (see subsection on the construct properties for the + properties).

B4) **fix** □: funct(behavior,behavior) → behavior

fix *bhfunct* represents a process whose activity is the same as *bhfunct*(**fix** *bhfunct*). This operator allows one to represent processes with nonterminating activities. For example, **fix** λx. a Δ x represents the process which goes on forever performing the action a. It is important to note that the **fix** operator is total and that the above operational characterization allows one to define completely the processes represented by it. For example, **fix** λx. x is defined and represents the process unable to perform any activity, which is also indicated by **stop.**

B5) □ ; □: behavior × behavior → behavior

Sequential composition of behaviors: The activity of bh_1; bh_2 consists of the activity of bh_1 until it terminates, followed by the activity of bh_2, if bh_1 has terminated correctly (i.e., in the state represented by **skip**). **skip** is a left identity for ;, while **stop** is a kind of zero for ; (stop ; bh is a behavior unable to perform any activity).

B6) **trap** □ **in** □: map(eid,behavior) → behavior

B7) **exit** □:eid → behavior

where map(eid,behavior) is the type of the finite mappings from the CL exception identifiers into behaviors.

The activity of **trap** *emap* **in** bh consists of the activity of bh; moreover, if bh terminates performing an **exit** to an exception ei and ei belongs to the domain of *emap*, then the activity goes on as specified by *emap*(ei); otherwise, the **exit** is propagated to some outer **trap** operator.

We also need another kind of sequential composition for behaviors—more precisely, the possibility of composing a behavior which terminates its executions returning a value of a certain type, with another behavior waiting for a value of the same type. The behaviors returning a value of type arg are elements of type arg-behavior, while the behaviors waiting for a value of type arg are represented by functions from arg into behaviors [elements of type funct(arg,behavior)].

For arg = val, env;

B8) **def**$_{arg}$ □ **in** □:
arg-behavior × funct(arg,behavior) → behavior

B9) **return**$_{arg}$ □:arg → arg-behavior

The activity of **def**$_{arg}$$bh$ **in** *bhfunct* consists of the activity of bh until it terminates, followed by the activity of *bhfunct*(a) if bh terminates returning the element of type arg a; **return**$_{arg}$$a$ represents the final state of a process which has terminated its activity returning a.

The elements of type env = map(vid,loc) (for environment) are just finite mappings from CL variable identifiers into locations of the shared storage; for the type env there are the same operators as for the type store, introduced in the global information part.

For the elements of type arg-behavior, ***SYST*** has the operators Δ, **choose**, **if** □ **then** □ **else** □, **def**, similar to those for the elements of type behavior.

Parallel operator

The elements of type state represent the states of the concurrent system SYST:

- ⟨□,□⟩: mset(behavior) × store → state

The elements of type mset(behavior) are finite multisets of behaviors and we choose to represent a multiset, whose elements are $bh_1 \cdots bh_n$, as $bh_1 | \cdots | bh_n$ to suggest that the processes represented by the various behaviors are in parallel.

In a state having form ⟨$bh_1 | \cdots | bh_n$,sh⟩ the processes represented by $bh_1 \cdots bh_n$ can perform their actions freely; i.e., either they can act (if possible) or can stay lazy. The only restriction is on the access to the shared store: Two contemporaneous updatings of the same cell are not allowed.

Semantics of the intermediate language The semantics of the intermediate language is given, specifying the abstract data types labeled transition systems associated with behaviors and with the concurrent system of behaviors in parallel, and then defining an observational semantics (i.e., a semantic equivalence on the labeled trees). All this is made precise in the following section.

• *Formal definition of the intermediate language*

The intermediate language ***SYST*** is formally defined by describing the concurrent system SYST as an abstract data type and giving it an observational semantics; to do this we use some classic algebraic techniques. In the appendix we collect the basic notions needed for understanding the paper.

As pointed out in the introduction, the use of algebraic techniques has a variety of motivations: First, data types of any complexity can be specified modularly within the same abstract framework; second, various kinds of semantics can be expressed using powerful classical methods (initial and terminal algebra semantics); third, rapid prototyping tools can be developed for symbolic execution (indeed, we have a SMoLCS interpreter).

Overall structure

As said before, the programs of ***SYST*** represent the states of a concurrent labeled transition system SYST; here we define this system, following the SMoLCS methodology (see [12–14]), as an abstract data type by an algebraic specification (see, e.g., [19]). This specification is based on the specifications STATE with sort state (the states of the system) and FLAG with sort flag [the flags (labels) of the system], and has a ternary Boolean operation symbol (the transition relation)

$$\Box \xrightarrow{\Box} \Box\text{: state} \times \text{flag} \times \text{state} \rightarrow \text{bool},$$

which is defined by a set of positive conditional axioms having form

$$\text{cond} \supset s \xrightarrow{f} s' = \text{True}.$$

In what follows we will write $s \xrightarrow{f} s'$ instead of $s \xrightarrow{f} s' = \text{True}$.

Formally this can be presented as the following specification schema:

enrich STATE + FLAG + BOOL by

opns $\Box \xrightarrow{\Box} \Box$: state × flag × state → bool

axioms "axioms defining →"

(BOOL is a specification of truth values).

Above we have used the operator "+," which builds the sum of two specifications T+T′. Formally the signature of T+T′ is the union of the signatures of T and T′, and the set of axioms of T+T′ is the union of the sets of axioms of T and T′. The other operator

enrich T by sorts S opns O axioms A,

roughly similar to the enrich-operation of CLEAR (see [20]), can be derived by using "+." It corresponds to

T + (sorts (S ∪ Sorts(T)) opns (O ∪ Opns(T)) axioms A);

whenever the parts sorts and/or opns and/or axioms are lacking, S and/or O and/or A are considered to be equal to ∅.

The specification SYST can be seen as an instantiation of a schema shown elsewhere [12–14]; here for simplicity we define it directly.

The system SYST is specified in four steps, each one consisting of the algebraic specification of a labeled transition system: BH-SYST, S-SYST, P-SYST, and finally SYST.

BH-SYST (behavior system) is a labeled transition system where states (terms of sort behavior) and transitions correspond to the states and the basic capabilities of actions of the subcomponent processes of SYST.

The states of S-SYST (S stands for synchronization) are parallel compositions of states of subcomponent processes (multisets of), plus some global information (states of the shared store). The transitions are the results of synchronized cooperation among subcomponent processes. Note that, for technical convenience, single process transitions are embedded into the new ones; moreover, creation of a new process is seen as synchronization of the creating process with a process represented by **seed** evolving into the created one performing an action START(···) [**seed** is a special behavior, whose only actions have form START(···), and a state of S-SYST can include any number of **seed**].

The transitions of P-SYST (P stands for parallelism) correspond to contemporaneous (parallel) executions of several synchronous actions. Here the states are the same as for S-SYST, and we take care of the mutually exclusive access to the shared store.

In the general SMoLCS schema SYST corresponds to the monitoring step, where the states are the same as those of P-SYST. Here the global constraints imposed by the monitoring say only that the duration of a task action can be anything.

Formal specification

In what follows, VAL, EID, LOC, and CHID are algebraic specifications which are not further specified (VAL also includes the Boolean values).

The behavior system BH-SYST The actions of the behaviors are defined by the following specification:

ACT = enrich VAL + LOC + CHID by

sorts act, behavior

opns START: behavior → act

$A_1 \cdots A_5$

$A_1 \cdots A_5$ are the operators for expressing behavior actions introduced in the previous subsection, and START represents the actions of **seed.**

For giving a semantics following the direct style, we need to define behaviors returning values and environments; so we need to define algebraically the environments:

ENV = MAP(VID,LOC)[env/map(vid,loc)]

The notation $A[srt_1/srt_2]$ means that in the specification A the sort srt_2 is renamed srt_1. The states of BH-SYST are defined by the following specification:

BEHAVIOR =

enrich ACT + MAP(EID,BEHAVIOR⟨behavior⟩) +
FUNCT(BEHAVIOR⟨behavior⟩,
BEHAVIOR⟨behavior⟩) +
ARG-FUNCT by
sorts behavior, val-behavior, env-behavior
opns **seed:** → behavior
$B_0 \cdots B_9$

where ARG-FUNCT =
\+ FUNCT(S,BEHAVIOR⟨bh⟩)
S=VAL,ENV,LOC,BOOL
bh=behavior,val-behavior,env-behavior

and $B_0 \cdots B_9$ are the ***SYST*** operators for expressing behaviors introduced in the previous subsection. Given a specification A and one of its sort srt, $A\langle srt\rangle$ means that srt is chosen as the main sort of A. MAP is the parametric algebraic specification of finite maps (for a complete definition, see, e.g., [14]). Given two algebraic specifications A and B with main sorts a and b, respectively, FUNCT(A,B) indicates the specification, with sort funct(a,b), of the functions from elements of sort a into elements of sort b; FUNCT(A,B) has the operators F1, F2, F3 introduced in the previous section (see [8,21]).

Note that BEHAVIOR is defined in a recursive way. Indicating by **BH** the transformation implicitly defined by the right-hand side of the definition, we can write BEHAVIOR = **BH**(BEHAVIOR). It is easy to check that, defining BEHAVIOR$^{\circ}$ as the specification with just the sorts behavior and t-behavior for t = val, env, and neither operations nor axioms, and BEHAVIOR^{n} = **BH**(BEHAVIOR^{n-1}), for n=3, we get the fixpoint specification BEHAVIOR.

BH-SYST is defined by enriching BEHAVIOR with the operations

$\square \xrightarrow{\square} \square$: behavior × act × behavior → bool,
and for t = val, env

$\square \xrightarrow{\square} t\rangle \square$: t-behavior × act × t-behavior → bool

(−t> represents the transition relation for the behaviors returning elements of sort t) defined by the following axioms, which formally reflect the semantics of the ***SYST*** operators informally introduced in the previous subsection.

- Creation of a new behavior

  seed $\xrightarrow{\text{START(bh)}}$ bh

- Usual operational definition of **fix** operator

  fix bhfunct = bhfunct(**fix** bhfunct)

- The following axioms define the **trap/exit** operators. The activity of **trap** *emap* **in** *bh* consists of the activity of *bh* (1, 2); moreover, if *bh* terminates performing an **exit** to an exception *ei* and *ei* belongs to the domain of *emap*, then the activity goes on as specified by *emap*(*ei*) (3); otherwise, the **exit** is propagated to some outer **trap** operator (4).

  (1) bh $\xrightarrow{a}$ bh′ ⊃ **trap** emap **in** bh $\xrightarrow{a}$ **trap** emap **in** bh′

  (2) **trap** emap **in skip** = **skip**

  (3) ei ∈ Dom(emap) = True ⊃
  trap emap **in exit** ei = emap(ei)

  (4) ei ∈ Dom(emap) = False ⊃
  trap emap **in exit** ei = **exit ei**

- Conditional

  if True **then** bh_1 **else** bh_2 = bh_1

  if False **then** bh_1 **else** bh_2 = bh_2

- Action prefixing

  a Δ bh $\xrightarrow{a}$ bh

- Sequential composition

  bh $\xrightarrow{a}$ bh′ ⊃ bh;bh_1 $\xrightarrow{a}$ bh′;bh_1

  skip;bh = bh

  exit ei;bh = **exit** ei

- choose operators
  For arg = val, loc, bool;

  arg-funct(x-arg) $\xrightarrow{a}$ bh′ ⊃ $\mathbf{choose}_{arg}$ arg-funct $\xrightarrow{a}$ bh′

  Isfree(v-arg,emap) = False ⊃

  trap emap **in** $\mathbf{choose}_{arg}\lambda$v-arg.bh =
  $\mathbf{choose}_{arg}\lambda$v-arg.**trap** emap **in** bh

  Isfree(v-arg,emap) = False ⊃

  ($\mathbf{choose}_{arg}\lambda$v-arg.bh);$bh_1$ = $\mathbf{choose}_{arg}\lambda$v-arg.(bh;$bh_1$)

  For arg1 = val,env;

Isfree(v-arg,arg1-funct) = False ⊃

$\mathbf{def}_{arg1}$(**choose**$_{arg}$λv-arg.arg1-bh) **in** arg1-funct =

choose$_{arg}$λv-arg.(**def**$_{arg1}$arg1-bh **in** arg1-funct)

- **def/return** operators

For arg = val, env;

arg-bh arg> arg-bh′ ⊃

def$_{arg}$ arg-bh **in** arg-funct

def$_{arg}$ arg-bh′ **in** arg-funct

def$_{arg}$ **return**$_{arg}$ x-arg **in** arg-funct = arg-funct(x-arg)

Here we have chosen the names of the variables appearing in the axioms in a way which recalls their sorts; e.g., bh is a variable of sort behavior, arg-bh of sort arg-behavior, x-arg of sort arg, arg-funct of sort funct(arg,behavior), bhfunct of sort funct(behavior,behavior), and so on.

Isfree is a total operation of the parametric specification FUNCT such that Isfree(v,x) is equal to True iff the "variable" v appears in x not enclosed by an operator λv. · · · .

For lack of room we have not reported here the axioms relative to the operators Δ, **choose, if** □ **then** □ **else** □, **def**$_{arg}$ for the elements of val-behavior and env-behavior; they are analogous to the ones for the elements of sort behavior defined above (e.g., the axioms for Δ are

a Δ arg-bh $\xrightarrow{a}$ arg> arg-bh and a Δ env-bh $\xrightarrow{a}$ env> env-bh).

The synchronous system S-SYST The states of S-SYST are couples whose components are a multiset of behaviors (states of the component processes) and a state of the shared store; they are defined by the following specification:

STATE = enrich PROD(MSET(BEHAVIOR),STORE)

[state/prod(mset(behavior),store)] by

axioms ⟨**seed**|bms,sh⟩ = ⟨bms,sh⟩

STORE = MAP(LOC,VAL)[store/map(loc,val)]

PROD and MSET indicate the parametric algebraic specifications of Cartesian product and finite multiset, where ⟨□,□⟩ is the couples constructor, | indicates multiset union, and the singleton multiset {e} is simply written e.

The added axiom formalizes the fact that a state of S-SYST includes whatever number of instances of **seed** and then allows one to handle dynamic creation of new processes.

As flags of the synchronous actions we simply choose the same flags as the behavior system (elements of sort act).

S-SYST is given by enriching STATE + BH-SYST with the operation □ = □ => □: state × act × state → bool, defined by the following axioms:

- Process internal action

$bh \xrightarrow{TAU} bh' \supset \langle bh,sh\rangle = TAU => \langle bh',sh\rangle$

- Allocation of an unused cell of the shared store (Undef is a zero-ary operation of VAL)

$bh \xrightarrow{ALLOC(l)} bh' \wedge l \in Dom(sh) = False \supset$

$\langle bh,sh\rangle = WRITE(l,Undef) => \langle bh',sh[Undef/l]\rangle$

- Reading and writing a cell of the shared store

$bh \xrightarrow{READ(l,v)} bh' \wedge l \in Dom(sh) = True \wedge sh(l) = v \supset$

$\langle bh,sh\rangle = TAU => \langle bh',sh\rangle$

$bh \xrightarrow{WRITE(l,v)} bh' \wedge l \in Dom(sh) = True \supset$

$\langle bh,sh\rangle = WRITE(l,v) => \langle bh',sh[v/l]\rangle$

- Handshaking communication

$bh_1 \xrightarrow{SEND(cid,v)} bh_1' \wedge bh_2 \xrightarrow{REC(cid,v)} bh_2' \supset$

$\langle bh_1|bh_2,sh\rangle = TAU => \langle bh_1'|bh_2',sh\rangle$

- Creation of a new process

$bh \xrightarrow{CREAT(bh_1)} bh' \wedge \mathbf{seed} \xrightarrow{START(bh_1)} bh_1 \supset$

$\langle bh|\mathbf{seed},sh\rangle = TAU => \langle bh'|bh_1,sh\rangle$

The parallel system P-SYST The flags of P-SYST are defined by the following specification:

PFLAG = enrich ACT by

opns □//□: act × act → act

Writing: loc × act → bool

axioms $a_1 // a_2 = a_2 // a_1$

$(a_1 // a_2) // a_3 = a_1 // (a_2 // a_3)$

$Writing(l,TAU) = False$

$Writing(l_1,WRITE(l_2,v)) = Equal(l_1,l_2)$

$Writing(l,a_1 // a_2) =$

$Writing(l,a_1) \vee Writing(l,a_2)$

The operation // defines the flags of the parallel actions corresponding to the contemporaneous execution of several synchronous actions; Writing(l,a) is True iff a is the flag of a parallel action in which the location l is written. [Note that there are no synchronous actions labeled by READ(l,v) or ALLOC(l)].

The states of P-SYST are the same as S-SYST and the transition relation of P-SYST is an extension of the one for S-SYST:

P-SYST =

enrich S-SYST + PFLAG by

axioms

$\langle bms_1,sh\rangle$ =a=> $\langle bms_1',sh'\rangle$ ∧

$\langle bms_2,sh\rangle$ =TAU=> $\langle bms_2',sh\rangle$ ⊃

$\langle bms_1|bms_2,sh\rangle$ =a//TAU=> $\langle bms_1'|bms_2',sh'\rangle$

$\langle bms_1,sh\rangle$ =a=> $\langle bms_1',sh'\rangle$ ∧

$\langle bms_2,sh\rangle$=WRITE(l,v)=>$\langle bms_2',sh[v/l]\rangle$ ∧

Writing(l,a) = False ⊃

$\langle bms_1|bms_2,sh\rangle$ =a//WRITE(l,v)=>

$\langle bms_1'|bms_2',sh'[v/l]\rangle$.

Note how the above axioms formalize the constraint, informally introduced in the previous subsection, that two contemporaneous updatings of a single cell of the shared store are not allowed. Note that the final state of the shared store does not depend on the order in which the synchronous actions are composed.

The overall system SYST As the whole system is closed, i.e., there are no interactions with the external world, the flags of the system SYST are simply defined by FLAG = sorts flag opns TAU:⟶ flag. The states of SYST are the same as P-SYST, and the transition relation of SYST is indicated by ===>.

SYST = enrich P-SYST + FLAG by

opns □==□==>□: state × flag × state ⟶ bool

axioms $\langle bms,sh\rangle$ =a=> $\langle bms',sh'\rangle$ ⊃

$\langle bms|bms_1,sh\rangle$==TAU==>$\langle bms'|bms_1,sh'\rangle$.

The axiom says that any group of processes (bms_1) can always wait, formalizing the fact that the duration of the process actions can be anything, and moreover that any action allowed in P-SYST (the premise of the axiom) is allowed to happen in an overall transition.

Semantics

Operational semantics It is interesting to show that choosing an initial algebra approach for the semantics of the algebraic specification of a labeled transition system allows one to define an operational semantics. Indeed, we have the following result.

Consider the following equivalence on the terms built on the signature of SYST for which the definedness can be proved:

$t_1 \equiv_I t_2$ iff SYST $\vdash t_1 = t_2$.

It can be shown that the equivalence $\equiv_I$ is a congruence; i.e., it is compatible with the operations, hence we can define a partial algebra I_{SYST} where the objects are the equivalence classes.

Proposition 1 The algebra I_{SYST} is initial in the class of partial models of SYST and is such that

I1 $I_{SYST} \models \mathbf{D}(t)$ iff SYST $\vdash \mathbf{D}(t)$;

I2 SYST $\vdash \mathbf{D}(t_1) \wedge \mathbf{D}(t_2)$ implies

$(I_{SYST} \models t_1 = t_2$ iff SYST $\vdash t_1 = t_2)$

(see the appendix for the definition of **D**).

Proof Initiality follows, e.g., from [15,22]; properties I1 and I2 are shown easily by the definition of I_{SYST}. □

It is clear that the definition of I_{SYST} allows one to define labeled trees associated with equivalence classes of states. Then it is possible to apply to I_{SYST} the usual techniques for defining various kinds of operational semantics; for example, we can consider two states (classes) equivalent iff their associated trees are the same.

However, operational semantics is not sufficient for expressing the real meaning, since it discriminates too much. For example, neither $\langle$TAU Δ TAU Δ **skip**,*sh*$\rangle$, $\langle$TAU Δ **skip**,*sh*$\rangle$ nor $\langle$WRITE(*l*,*v*) Δ **skip**,*sh*$\rangle$, $\langle$WRITE(*l*,*v*) Δ WRITE(*l*,*v*) Δ **skip**,*sh*$\rangle$ are couples of operationally equivalent states of SYST, contrary to our intuition of their intended meaning.

Hence, as we have already discussed in Section 3, we need to consider two objects equivalent up to some observations. Formally we will define classes of observational semantics.

Observational semantics As already emphasized, we are able to accommodate various kinds of observational semantics. Let us assume, just for the sake of example, that in the case of SYST we are interested in a result semantics; i.e., two states are considered equivalent iff they produce the same results, where the results of a state *s* are the final states of the shared store of the correctly terminating executions of *s* (i.e., where in the final state all behaviors have form **skip**); note that here we do not care about incorrect and nonterminating executions.

The paradigm under which an observational semantics is defined for a concurrent system (here applied to SYST) consists essentially of

- Giving a specification, defining the observations on the system (here SYST-PLUS), by means of Boolean relations (here Isres) stating that some observation values (here states of the shared store, defined by the specification

STORE) are true of some system states. Here we can define

SYST-PLUS = enrich SYST by

opns Isres: state × store → bool

axioms

Isres(⟨**skip**|···|**skip**,sh⟩,sh) = True

s ==TAU==>s′ ∧

Isres(s′,sh) = True ⊃

Isres(s,sh) = True.

- Defining a class of observationally equivalent algebras, each one containing the objects to be observed together with the relations and moreover preserving, as a subtype, a fixed model of the observed values (here the initial model of STORE).
- Defining the observational semantics as represented by a special algebra in the above class; formally it is the minimally defined and term-generated algebra terminal in that class (here CALG); a basic general theorem (in [15]) shows that this algebra has indeed the properties required of an observational semantics. Formally we get the following result.

Proposition 2 There exists an algebra CALG with the following properties.

For any srt ∈ Sorts(STATE)-Sorts(STORE) and for any ground terms of sort srt t,t' built on the signature of STATE,

O1 CALG ⊨ **D**(t) iff SYST ⊢ **D**(t)

O2 CALG ⊨ $t = t'$ iff

for any ground term sh of sort store,
for any context $s[x]$ built on the signature of STATE of sort state with a hole of sort srt

[SYST-PLUS ⊢ Isres($s[t]$,sh) = True iff

SYST-PLUS ⊢ Isres($s[t']$,sh) = True].

Proof Application of the main theorem in [15]. □

As before, for I_{SYST}, the elements of the algebra CALG are, up to isomorphism, congruence classes of terms, whose definedness can be proven in SYST.

Property O1 says that all the interesting objects of STATE are defined in CALG; by property O2 two terms of sort srt are equivalent if and only if in every context of sort state they satisfy the same observations. It is most important to note that in this way every subcomponent of a state gets an observational semantics: In SYST this is true, for example, of a behavior.

An important point to be understood here is why we need our observational semantics to be represented by an algebra, which is the same as requiring the equivalence induced by the observations to be a congruence. The semantic model of an abstract data type always has to be an algebra, in order for the operations to be interpreted on that model; in other words that means that the semantics is a homomorphism, which is nothing but the mathematical formal counterpart of the informal phrase "compositional semantics." It will be seen later that by using that homomorphism and composing it with the homomorphic translation of the first step, we can define the overall semantics of a language as a homomorphic mapping of the algebra of the syntax into a semantic algebra, which is the exact mathematical formalization of the phrase "denotational semantics."

- *Properties of the language constructs*

In this section we show some properties of the constructs of ***SYST***, which are the basis for proving properties of the system specifications.

These properties state the strong bisimulation (indicated by ~) in the sense of Milner (see [16]) of the processes represented by various behaviors. Roughly speaking, two behaviors (processes) are equivalent iff the associated labeled trees, where we observe only the arc labels and disregard the intermediate states (node labels), are the same. We are interested in properties of this kind because, given two states of a concurrent system specified following the SMoLCS methodology, s_1 and s_2, if s_1 and s_2 have the same information part and their component subsystems are pairwise equivalent with respect to ~, then s_1 and s_2 are observationally equivalent for all observational semantics which do not observe the states of the component subsystems, but at most their action capabilities. Thus, given two behaviors of SYST, bh_1 and bh_2, $bh_1 \sim bh_2$ implies CALG ⊨ $bh_1 = bh_2$.

Now, we formally define the strong bisimulation relation between behaviors.

In what follows we use bh and a to range over the behaviors and the actions of ***SYST***:

- $bh_1 \sim bh_2$ iff

  i) SYST ⊢ $bh_1 \xrightarrow{a_1} bh_1'$ implies there exist a_2, bh_2' such that

  SYST ⊢ $bh_2 \xrightarrow{a_2} bh_2' \wedge a_1 \sim a_2 \wedge bh_1' \sim bh_2'$.

  ii) SYST ⊢ $bh_2 \xrightarrow{a_2} bh_2'$ implies there exist a_1, bh_1' such that

  SYST ⊢ $bh_1 \xrightarrow{a_1} bh_1' \wedge a_1 \sim a_2 \wedge bh_1' \sim bh_2'$.

- $a_1 \sim a_2$ iff

($a_1 \neq$CREAT(···),$a_2 \neq$CREAT(···),$a_1 \neq$START(···),

$a_2 \neq \text{START}(\cdots)$ and SYST $\vdash a_1 = a_2$)

or ($a_1 = \text{CREAT}(bh_1)$, $a_2 = \text{CREAT}(bh_2)$ and $bh_1 \sim bh_2$)

or ($a_1 = \text{START}(bh_1)$, $a_2 = \text{START}(bh_2)$ and $bh_1 \sim bh_2$).

def/return *properties*
For arg, arg1 = val, env;

- $\textbf{def}_{arg}\ (a\ \Delta\ bh)$ **in** *arg-funct* $\sim a\ \Delta\ (\textbf{def}_{arg}\ bh$ **in** *arg-funct*)
- If x is not free in *arg-funct*, then

  $\textbf{def}_{arg}(\textbf{def}_{arg1}\ bh$ **in** $\lambda x.bh')$ **in** *arg-funct* $\sim$

  $\textbf{def}_{arg1}\ bh$ **in** $\lambda x.(\textbf{def}_{arg}\ bh'$ **in** *arg-funct*)
- If x is not free in bh'', then

  $(\textbf{def}_{arg}\ bh$ **in** $\lambda x.bh');bh'' \sim \textbf{def}_{arg}\ bh$ **in** $\lambda x.(bh';bh'')$.

fix property
A behavior of *bh* is said to be $\sim$-*image finite* iff for all actions a

$\{bh' \mid \text{SYST} \vdash bh \xrightarrow{a'} bh',\ a \sim a'\}$ is finite module **=** (the provable equality in SYST).

- If **fix** $\lambda x.bh_1$ and **fix** $\lambda x.bh_2$ are $\sim$-*image finite*, then

  [for all $n \geq 0$ $(\lambda x.bh_1)^n(\textbf{stop}) \sim (\lambda x.bh_2)^n(\textbf{stop})$] implies **fix** $\lambda x.bh_1 \sim$ **fix** $\lambda x.bh_2$

  ($f^0(a) = a, f^{n+1}(a) = f(f^n(a))$).

This property gives a very useful tool for proving the strong equivalence of behaviors expressed by means of the **fix** operator.

trap/exit *properties*
In the following we use *emap* to range over the elements of sort map(eid,behavior).

- **trap** *emap* **in** $(a\ \Delta\ bh) \sim a\ \Delta$ (**trap** *emap* **in** *bh*).
- For arg = val, env; if x is not free in *emap*, then

  trap *emap* **in** $(\textbf{def}_{arg}\ bh$ **in** $\lambda x.bh_1) \sim$

  $\textbf{def}_{arg}\ bh$ **in** $\lambda x.$(**trap** *emap* **in** bh_1).
- If *emap* does not include an **exit** to an exception *ei*, with $ei \in \text{Dom}(emap)$, then

  trap *emap* **in** $(bh_1;bh_2) \sim$

  trap *emap* **in** $(bh_1;$**trap** *emap* **in** $bh_2)$.

Let *emap*;*bh* indicate the element of sort map(eid,behavior) such that

Dom(*emap*;*bh*) = Dom(*emap*) and

(*emap* ; *bh*)(*ei*) = *emap*(*ei*);*bh*;

- If bh_1 does not include an **exit** to an exception *ei*, with $ei \in \text{Dom}(emap)$, then

  (**trap** *emap* **in** *bh*) ; $bh_1 \sim$ **trap** (*emap* ; bh_1) **in** $(bh;bh_1)$.

if then else *properties*

- (**if** *bv* **then** bh_1 **else** bh_2);$bh \sim$

  if *bv* **then** $(bh_1;bh)$ **else** $(bh_2;bh)$.
- **trap** *emap* **in** (**if** *bv* **then** bh_1 **else** $bh_2) \sim$

  if *bv* **then** (**trap** *emap* **in** bh_1) **else** (**trap** *emap* **in** bh_2).

; properties

- $bh_1;(bh_2;bh_3) \sim (bh_1;bh_2);bh_3$.
- $(a\ \Delta\ bh);bh_1 \sim a\ \Delta\ (bh;bh_1)$.
- stop;$bh \sim$ stop skip;$bh \sim bh$.

+ properties

- $bh_1 + (bh_2 + bh_3) \sim (bh_1+bh_2)+bh_3$.
- $bh_1+bh_2 \sim bh_2+bh_1$.

5. Formal semantics of concurrent languages

- *Semantic domains and functions*

Following the paradigm of denotational semantics, semantics is given as a many-sorted homomorphism from (abstract) syntax into a semantic algebra. In our approach some domains of the semantic algebra are defined as carriers of another algebra, the concurrent algebra CALG, which, as we have seen, is a model of an algebraic specification of a concurrent system whose states represent the execution states of the programs. Note that CALG is nothing but the semantics of the intermediate language, a semantics which varies depending on the observations we want to make.

Let us assume in the following, for the example language CL, that CALG is the concurrent algebra defined in the preceding section (and corresponding to a result semantics).

The semantic functions, i.e., the functions constituting the homomorphism, are

P: PROG	$\rightarrow$	*ANSWER*
S: STAT	$\rightarrow$	(*ENV* $\rightarrow$*BH*)
E: EXP	$\rightarrow$	(*ENV* $\rightarrow$*VAL-BH*)
D: DECS	$\rightarrow$	(*ENV* $\rightarrow$*ENV-BH*)
H: HANDLERS	$\rightarrow$	(*ENV* $\rightarrow$*HND*)

C: CONST $\rightarrow$ VAL
B: BOP $\rightarrow$ $((VAL \times VAL) \rightarrow VAL)$

The semantic domains above are defined as follows. If *srt* is a sort of a specification A, then A_{srt} indicates the carrier of sort *srt* in A.

From the definition of CALG it follows that a CL program has as value an element in $\text{CALG}_{\text{state}}$ representing the observational semantics of the program, i.e., an equivalence class of programs such that two programs are equivalent iff they produce the same observational results. Thus, we define

$$ANSWER = \text{CALG}_{\text{state}}.$$

Using the direct semantics, the meanings of statements will be elements of sort behavior in CALG, i.e., equivalence classes of behaviors, which are syntactic objects representing processes. Indeed, due to the presence of concurrency, statements are now modeled by processes:

$$BH = \text{CALG}_{\text{behavior}}$$

Also, the environment is defined as a subalgebra of CALG, because there are behaviors which terminate producing environments:

$$ENV = \text{CALG}_{\text{env}}$$

The meanings of expressions and declarations are still behaviors, but particular behaviors; the elements of *VAL-BH* are indeed behaviors, representing processes whose activities terminate returning a value, while the elements of *ENV-BH* are behaviors whose activities terminate returning an environment:

$$VAL\text{-}BH = \text{CALG}_{\text{val-behavior}}$$

$$ENV\text{-}BH = \text{CALG}_{\text{env-behavior}}$$

The storable values are defined by means of a subspecification (VAL with a sort val) of the concurrent systems representing program executions (SYST):

$$VAL = \text{CALG}_{\text{val}}$$

Moreover, from the definition of CALG (see the subsection "Formal specification") we have that $VAL = (\text{I}_{\text{VAL}})_{\text{val}}$, where I_{VAL} indicates the initial model of VAL (because VAL is a subspecification of STORE); thus we have that the values are abstractly specified.

The semantic value of a handler part of a block is a map from exception identifiers into statement values (behaviors):

$$HND = \text{CALG}_{\text{map(eid,behavior)}}$$

Note that there is a correspondence between the semantic domains used for sequential languages and those used for concurrent languages: The domain used in the sequential case ($STORE \rightarrow STORE$) corresponds to behaviors, while domains having form ($STORE \rightarrow (STORE \times ABC)$) correspond to behaviors terminating producing elements of ABC.

• *Semantic clauses*

The definition of the semantic functions is by structural induction on the syntactic structure. It is important to note that we consider only statically correct programs; moreover, for simplicity we do not make provision for run-time error checks; i.e., error messages are just values in the appropriate domains.

Notice that, as we have often emphasized, the clauses can be seen as defining a syntax-directed translation into the intermediate language. The overall semantics is then obtained by interpreting in CALG the terms corresponding to language constructs (programs, statements, declarations, and expressions). The full clauses are given in **Figure 3**; here we provide some comments.

Clause 1 defines the semantics of a CL program consisting of the tasks $bl_1 \cdots bl_n$ as the value in the concurrent algebra CALG of the term $\langle \text{S}[bl_1]\rho_o | \cdots | \text{S}[bl_n]\rho_o, \text{sh}_o \rangle$, that is, the syntactic representation of a state of the labeled transition system SYST (defined in the subsection "The overall system SYST" as an algebraic specification) representing the executions of the CL programs. Formally $\langle \text{S}[bl_1]\rho_o | \cdots | \text{S}[bl_n]\rho_o, \text{sh}_o \rangle$ is a term of sort state in the specification STATE, subspecification of SYST; its semantic value, as already mentioned, is an equivalence class corresponding to an observational semantics on SYST. For ease of notation here and in the following we simply indicate by t the semantic value (interpretation) t^A of a term t in an algebra A.

A state of SYST has two components: The first is a multiset of behaviors (corresponding to CL tasks), while the second represents some information global to all behaviors (the state of the shared store). In clause 1 ρ_o represents the initial environment and sh_o the initial state of the shared store, and in clause 2 ρ is a generic element in ENV.

The behavior associated by **E** with an expression has in some way the possibility of returning a value, while the behavior operator $\textbf{def}_{\text{val}}$...**in**... composes a behavior returning a value with a behavior waiting for a value, producing another behavior. The behavior waiting for a value is defined by the term of sort funct(val,behavior) $\lambda v.\ \text{WRITE}(\rho(x),v)\ \Delta$ **skip**, which represents a function from values to behaviors. Note that here $\lambda\square.\square$ is an operation of the specification STATE (see the subsection on the formal definition of ***SYST***) and it does not mean λ-notation, as in Section 2.

Clause 3 significantly looks the same as in the purely sequential case, as does clause 5. Note in clause 5 that ; is an operation on behaviors and in clauses 3 and 4 note also that **if**...**then**...**else**... is an operation on behaviors, and most

1 $\mathbf{P}[\underline{\text{program}}\ bl_1 \parallel \cdots \parallel bl_n] = \langle \mathbf{S}[bl_1]\rho_o | \cdots | \mathbf{S}[bl_n]\rho_o, sh_o \rangle$

2 $\mathbf{S}[x := e]\rho = \mathbf{def}_{val}\ \mathbf{E}[e]\rho\ \mathbf{in}\ \lambda v.\text{WRITE}(\rho(x),v)\ \Delta\ \mathbf{skip}$

3 $\mathbf{S}\ [\underline{\text{if}}\ be\ \underline{\text{then}}\ st_1\ \underline{\text{else}}\ st_2\ \underline{\text{fi}}]\rho = \mathbf{def}_{val}\ \mathbf{E}[be]\rho\ \mathbf{in}\ \lambda bv.\mathbf{if}\ bv\ \mathbf{then}\ \mathbf{S}[st_1]\rho\ \mathbf{else}\ \mathbf{S}[st_2]\rho$

4 $\mathbf{S}[\underline{\text{while}}\ be\ \underline{\text{do}}\ st\ \underline{\text{od}}]\rho = \mathbf{fix}\ \lambda bh.\mathbf{def}_{val}\ \mathbf{E}[be]\rho\ \mathbf{in}\ \lambda bv.\mathbf{if}\ bv\ \mathbf{then}\ (\mathbf{S}[st]\rho;bh)\ \mathbf{else}\ \mathbf{skip}$

5 $\mathbf{S}[st_1; st_2]\rho = \mathbf{S}[st_1]\rho;\mathbf{S}[st_2]\rho$

6 $\mathbf{S}[\underline{\text{raise}}\ ei]\rho = \text{TAU}\ \Delta\ \mathbf{exit}\ ei$

7 $\mathbf{S}[dcs\ \underline{\text{begin}}\ st\ hnd\ \underline{\text{end}}]\rho = \mathbf{def}_{env}\ \mathbf{D}[dcs]\rho\ \mathbf{in}\ \lambda\ \rho'.\mathbf{trap}\ \mathbf{H}[hnd]\rho'\ \mathbf{in}\ \mathbf{S}[st]\rho'$

8 $\mathbf{S}[\underline{\text{send}}(ci,e)]\rho = \mathbf{def}_{val}\ \mathbf{E}[e]\rho\ \mathbf{in}\ \lambda v.\text{SEND}(ci,v)\ \Delta\ \mathbf{skip}$

9 $\mathbf{S}[\underline{\text{rec}}(ci,x)]\rho = \mathbf{choose}_{val}\lambda v.\text{REC}(ci,v)\ \Delta\ \text{WRITE}(\rho(x),v)\ \Delta\ \mathbf{skip}$

10 $\mathbf{S}[\underline{\text{create task}}\ bl]\rho = \text{CREAT}(\mathbf{S}[bl]\rho)\ \Delta\ \mathbf{skip}$

11 $\mathbf{S}[st_1\ \underline{\text{or}}\ st_2]\rho = \mathbf{S}[st_1]\rho + \mathbf{S}[st_2]\rho$

12 $\mathbf{E}[c]\rho = \text{TAU}\ \Delta\ \mathbf{return}_{val}\ \mathbf{C}[c]$

13 $\mathbf{E}[bop(e_1,e_2)]\rho = \mathbf{def}_{val}\ \mathbf{E}[e_1]\rho\ \mathbf{in}\ \lambda v_1.\mathbf{def}_{val}\ \mathbf{E}[e_2]\rho\ \mathbf{in}\ \lambda v_2.\text{TAU}\ \Delta\ \mathbf{return}_{val}\ \mathbf{B}[bop](v_1,v_2)$

14 $\mathbf{E}[x]\rho = \mathbf{choose}_{val}\ \lambda v.\text{READ}(\rho(x),v)\ \Delta\ \mathbf{return}_{val}\ v$

15 $\mathbf{D}[\Lambda]\rho = \mathbf{return}_{env}\ \rho$

16 $\mathbf{D}[\underline{\text{var}}\ x\ dcs]\rho = \mathbf{choose}_{loc}\ \lambda\ l.\text{ALLOC}(1)\ \Delta\ \mathbf{D}[dcs](\rho[l/x])$

17 $\mathbf{H}[\Lambda]\rho = hnd_o$

$hnd_o = []$ ([] indicates the empty map)

18 $\mathbf{H}[\underline{\text{when}}\ ei\ \underline{\text{do}}\ st\ hnd]\rho = (\mathbf{H}[hnd]\rho)[\mathbf{S}[st]\rho/ei]$

Figure 3

Semantic clauses.

importantly in clause 4 note how fixpoints are handled. The actions of the behavior **fix** λbh. ··· are defined in the algebraic specification BH-SYST (subspecification of SYST) in a way corresponding to the usual rewriting rule for the operational semantics of fixpoint operators (see [10, 16]).

Clauses 6, 7, 17, and 18 explain how exceptions are handled. Thus, in handling exceptions our approach is also similar to that of the classical VDM denotational semantics, but now statement values are process (behavior) elements of *BH*. $\mathbf{H}[hnd]\rho'$, the meaning of the handlers part of the block, represents a map from exception identifiers into behaviors [a term of sort map(eid,behavior) in the specification STATE]; **trap**...**in**... and **exit**... are operators on behaviors.

Clause 14 makes explicit some hidden concurrency related to the evaluation of a variable. The idea behind the definition of the meaning of the expression x as a nondeterministic behavior is that the content of the location $\rho(x)$ will depend on the moment when the task within which the evaluation of x is performed gets access to the shared store; if v_o is the value in $\rho(x)$ at that moment, then the action READ$(\rho(x),v_o)$ will be performed and the value v_o will be returned. Analogously this is so for clauses 9 and 16.

Clause 12 (and 13) shows a typical situation: It would be as in the purely sequential case, but in the concurrent case an action is performed, consisting in evaluating the constant; this action is internal, i.e., does not involve either other processes or the shared store, and hence it is indicated by TAU. Representing that action is not necessary whenever the observational semantics on SYST only takes care of the action results and not of their ordering. On the other hand, in clauses 15 and 17 there is no TAU action, because Λ just means absence of, respectively, declarative part and handlers part.

Clauses 8, 9, 10, and 11 are the usual clauses for concurrent statements; the meaning of the behavior actions SEND, REC, CREAT is given by the definition of the synchronization, parallelism, and monitoring steps of SYST (see the subsection on the formal definition of ***SYST***).

The consistency of denotational clauses can be checked in two steps: first, by proving that the semantic functions are total on statically correct programs; i.e., they associate with every correct CL program a term on the signature of the specification STATE; second, by showing that all the terms

associated with such programs have a defined interpretation in the concurrent algebra CALG.

Proposition 3 For every correct $pr \in$ PROG

$\mathbf{P}[pr] \in \{ s \mid s$ term of sort state built on Sig(STATE) $\}$ and

$\text{CALG} \vDash \mathbf{D}(\mathbf{P}[pr])$.

Proof By analogous properties of the semantic functions **S**, **E**, **D**, **H**, **C**, **B** proved by induction on the syntactic structure of CL. □

Here we show by an example how our definition works.

Let $\underline{l}$ be an element of CONST and $\underline{\text{plus}}$ be an element of BOP, whose intuitive meanings are clear. In what follows we report the semantics of the CL statement $x := \underline{\text{plus}}(x,\underline{l})$:

$\mathbf{S}[x := \underline{\text{plus}}(x,\underline{l})]\rho =$

$\mathbf{def}_{\text{val}}\ \mathbf{E}[\underline{\text{plus}}(x,\underline{l})]\rho$

$\mathbf{in}\ \lambda \text{v}.\ \text{WRITE}(\rho(x),\text{v})\ \Delta\ \mathbf{skip} = (*)$

i) $\mathbf{E}[\underline{\text{plus}}(x,\underline{l})]\rho = \mathbf{def}_{\text{val}}\ \mathbf{E}[x]\rho$

$\mathbf{in}\ \lambda \text{v}_1.\ \mathbf{def}_{\text{val}}\ \mathbf{E}[\underline{l}]\rho$

$\mathbf{in}\ \lambda \text{v}_2.\text{TAU}\ \Delta$

$\mathbf{return}_{\text{val}}\ \mathbf{B}[\underline{\text{plus}}](\text{v}_1,\text{v}_2)$

ii) $\mathbf{E}[x]\rho = \mathbf{choose}_{\text{val}}\ \lambda \text{v}.\text{READ}(\rho(x),\text{v})\ \Delta\ \mathbf{return}_{\text{val}}\ \text{v}$

iii) $\mathbf{E}[\underline{l}]\rho = \text{TAU}\ \Delta\ \mathbf{return}_{\text{val}}\ l$

From i), ii), and iii) we have

$(*) = \mathbf{def}_{\text{val}}\ (\mathbf{def}_{\text{val}}(\mathbf{choose}_{\text{val}}\ \lambda \text{v}.\text{READ}(\rho(x),\text{v})\ \Delta$

$\mathbf{return}_{\text{val}}\ \text{v})$

$\mathbf{in}\ \lambda \text{v}_1.\ \mathbf{def}_{\text{val}}(\text{TAU}\ \Delta\ \mathbf{return}_{\text{val}}\ l)$

$\mathbf{in}\ \lambda \text{v}_2.\text{TAU}\ \Delta\ \mathbf{return}_{\text{val}}\ \text{v}_1+\text{v}_2)$

$\mathbf{in}\ \lambda \text{v}.\ \text{WRITE}(\rho(x),\text{v})\ \Delta\ \mathbf{skip}.$

By the properties of the ***SYST*** constructs listed at the end of the previous section, we can reduce the above behavior to a simpler form; then

$\mathbf{S}[x := \underline{\text{plus}}(x,\underline{l})]\rho =$

$\mathbf{choose}_{\text{val}}\ \lambda\ \text{v}.\ \text{READ}(\rho(\text{x}),\text{v})\ \Delta\ \text{TAU}\ \Delta\ \text{TAU}\ \Delta$

$\text{WRITE}(\rho(\text{x}),\text{v}+1)\ \Delta\ \mathbf{skip}.$

Conclusion

We have shown how the denotational approach to semantics using a direct (VDM-like) style can be extended to handle concurrent languages. Together with demonstrating this possibility, there are some other major novelties in our approach, namely the possibility of keeping an applicative style (denotational clauses) in an overall algebraic setting; the specification of concurrent processes as abstract data types; and finally the use of a highly modular and parameterized schema for expressing the semantics of concurrent systems.

There are some points we would like to emphasize, perhaps once more.

It should be clear first of all that the approach we have shown here on a toy example language can be applied to languages of any scale (and indeed it has been applied to giving the dynamic semantics of full Ada).

A most important point to bear in mind is that the overall apparatus consisting of the language of behaviors and of their parallel composition can be fixed once (see especially [9] on this point). Thus we can obtain a metalanguage for defining semantics, and if we adopt this metalanguage, for example, the semantics of CL becomes remarkably concise, reducing to denotational clauses. Clearly, in order to use with confidence such a metalanguage, more work has to be done to reach a kind of standardization, endowed with a set of derived properties of its operators, such as those presented in the section on the intermediate language, and some work is going on in that direction (see, e.g., [9]).

Moreover, we need automatic tools for ensuring correctness of the definition, if the scale of the language is rather large. To this end a system has been developed [17] which, starting from the denotational clauses, translates a program into the intermediate language and then can give the transitions of a state of a concurrent system specified in the SMoLCS style. This system is the analog for concurrent languages of a typical rapid prototyping interpreter for sequential languages.

Finally, we would like to mention that our approach permits proofs of interesting properties of semantic specifications, using properties such as those of behaviors presented in the section on the intermediate language. Indeed we can prove (see for an abstract of the results [23]) that the direct semantics style shown here and the continuation style used in [7] are equivalent in a very deep sense, since, for example, the two behaviors corresponding in the two styles to a statement are strongly equivalent in the sense we have seen. Moreover, it will be shown in a forthcoming paper that our semantics reduces to classical functional semantics on the purely sequential subset of a concurrent language; because of the result on the equivalence of direct and continuation semantics, this result applies to both styles.

Appendix: Basic notions on partial abstract data types

We give here a rather informal presentation of the key concepts, mainly in order to introduce the notational conventions for a reader not acquainted with algebraic specifications; a reader interested in formal aspects can refer to [22]. The basic idea is that a data type is not defined directly in a constructive way, but some properties are

indicated that the related operations must satisfy. The advantage of that approach is that we obtain an abstract description of a data type (a family of domains together with a set of functions which have arguments and results in these domains), i.e., what is called an *abstract data type.* Assume, for example, that we want to specify formally the well-known data type "stack." A concrete way of defining that data type is to see it, for example, as composed by the three sets

ELEM (the set of the elements which are put in the stack)
STACK = ELEM* (strings of elements)
BOOL = {true, false}

and by the functions

- Empty: → STACK
  Empty = Λ
- Isempty: STACK → BOOL
  Isempty(Λ) = true
  Isempty(e&s) = false
- Push: ELEM × STACK → STACK
  Push(e,s) = e&s
- Pop: STACK → STACK
  Pop(e&s) = s
- Top: STACK → ELEM
  Top(e&s) = e

where & and Λ indicate, respectively, string concatenation and empty string.

Note that Top and Pop are partial functions since Top(Λ) and Pop(Λ) are not defined.

That definition of stack corresponds to our intuition but seems in some sense too concrete in the sense that a particular model of the stack is chosen.

The following is a rather standard way of giving an "abstract" definition of stack, with a minor modification (the axioms with **D** are explained later) since here we use a partial algebra approach:

STACK = sorts elem, stack, bool
opns True, False: → bool
Empty: → stack
Pop: stack → stack
Push: elem × stack → stack
Pop: stack → elem
Isempty: stack → bool
axioms **D**(Empty) **D**(Push(e,s))
Top(Push(e,s)) = e
Pop(Push(e,s)) = s
Isempty(Λ) = True
Isempty(Push(e,s)) = False
"axioms expressing properties of booleans and stack elements"

The above definition is an example of *algebraic specification*, i.e., a couple consisting of a *signature* and a set of *axioms.*

A *signature* is a couple consisting of a set of symbols called *sorts* and a set of symbols called *operations.* Each operation has an associated *functionality* (a couple consisting of a string of sorts and a sort) and we write Op: s1 × ··· × sn → s to say that Op has functionality ⟨s1 ··· sn,s⟩.

For each signature we can consider the set of the expressions, usually called *terms* over the signature, inductively constructed by the operations, as follows:

i) For each zero-ary operation Op: → s, Op is a term of sort s;
ii) For each Op: s1 × ··· × sn → s, if t_1 ··· t_n are terms of sort, respectively, s1 ··· sn, then Op(t_1,···, t_n) is a term of sort s.

The intuitive meaning is that each term of sort s is a syntactic representation for an element in the set of values associated with s; for example, the term Pop(Push(e,Empty)) represents an element in the set associated with the sort stack, assuming that e is a term of sort element.

An *axiom*, as shown in the STACK specification, is a positive conditional formula between terms containing variables; i.e., its general form is

$$\bigwedge_i e_i \supset e,$$

where e_i is either $\mathbf{D}(t_i) \wedge t_i = t_i'$ or $\mathbf{D}(t_i)$ and e is either $\mathbf{D}(t)$ or $t = t'$ and t,t',t_i,t_i' are terms.

It should be clear that, because of the axioms, in general an element can be represented by many different terms; for example, Pop(Push(e,Empty)) and Empty represent the same element.

More precisely, we can consider equivalent all the terms t_1, t_2 whose equality can be proved using the axioms and the first-order logic; we write STACK $\vdash t_1 = t_2$ to indicate that t_1, t_2 can be proved equal in STACK.

For a given signature, fixing the set associated with each sort and interpreting each operation as a function (with the suitable domain and codomain accordingly with the functionality) gives a concrete data type (family of sets and functions) called an *algebra* on the signature. The value of a term obtained by assigning values to the variables and interpreting each operation as the corresponding function in the algebra is called the *interpretation* of the term in the algebra. We say that a data type is a *model* of the given algebraic specification iff with the given interpretation all the axioms hold; usually we write ALG $\models ax$ to indicate that the axiom ax holds in ALG.

As we have seen in the example of the stack, the interpretation of an operation can be a *partial function*; i.e., for some values of the arguments the result can be undefined, e.g., the interpretation of Pop is not defined on the empty stack.

Hence, in general, a model of a specification is a *partial data type* (or a *partial algebra*), i.e., a model where the

interpretation of an operation can be a partial function. Partiality has some important consequences:

1. The interpretation of the equality symbol = has to be defined with some care: In our approach $t_1 = t_2$ is satisfied in a model for some assignment of values to the variables iff the interpretations of t_1 and t_2 are either both defined and equal or both undefined.
2. We can speak of definedness of terms for some values of the variables in a model, but note that it is meaningless to speak of undefined values in a model. Hence an assignment of values to the variables can only be an assignment of defined values, and moreover the interpretation of the operations is *strict*; i.e., the interpretation of a term $Op(t_1,\cdots,t_n)$ is undefined whenever the interpretation of t_i for some i is undefined.
3. It is useful to specify the terms that we want to be defined in every model; that corresponds to requiring that some functions be defined in correspondence with certain argument values. Hence we use an overloaded symbol **D** to indicate the definedness predicates on terms, one for each sort; **D** is total in every model.

For example, in the specification of stack we have **D**(Empty) and **D**(Push(e,s)) to express that in every model Push is a total operation and Empty is defined.

In general, among all the models of an algebraic specification we can choose a particular model as "*the data type defined by the algebraic specification.*" A choice which is usually made is the model in which only the identifications which are forced by the axioms hold (i.e., the values of two terms are equal only if the two terms can be proved equal) and in which a term is defined iff its definedness can be proved from the axioms. That model is called the *initial model.* For algebraic specifications of transition systems also a different kind of model may be chosen, as is discussed at the end of the subsection on the formal definition of the intermediate language.

Acknowledgments

We gratefully acknowledge the benefits of many discussions and remarks resulting from the practical application of our approach to the Ada definition by our colleagues at the CRAI–Genoa group, A. Giovini, F. Mazzanti, and E. Zucca. Moreover, we warmly thank Ombretta Arvigo for patiently typing under pressure and at any time.

References

1. E. Astesiano, A. Giovini, F. Mazzanti, G. Reggio, and E. Zucca, "The Ada Challenge for New Formal Semantic Techniques," *Proceedings of the 1986 Ada International Conference*, Edinburgh, Scotland, Cambridge Press, 1986, pp. 239–248.
2. E. Astesiano and J. Storbank Pedersen, "An Introduction to the Draft Formal Definition of Ada," to appear in *Proceedings of the 3rd Workshop on Ada Verification*, Research Triangle Park, NC, 1986.
3. E. Astesiano, "Définition Sémantique Formelle du Langage Ada: Une Application de la Méthodologie SMoLCS en Logiciels," *Emergence des Normes, Enjeux* **75,** 37–39 (1986).
4. M. J. C. Gordon, *The Denotational Description of Programming Languages*, Springer-Verlag New York, 1979.
5. J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, London, 1977.
6. D. Bjørner and C. B. Jones, *Formal Specification and Software Development*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.
7. E. Astesiano and G. Reggio, "A Syntax-Directed Approach to the Semantics of Concurrent Languages," *Proceedings of the 10th IFIP World Congress* (H. J. Kugler, Ed.), North-Holland Publishing Co., Amsterdam, 1986, pp. 571–576.
8. E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini, P. Inverardi, E. Karlsen, F. Mazzanti, J. Storbank Pedersen, G. Reggio, and E. Zucca, *Deliverable 7 of the CEC MAP Project: The Draft Formal Definition of ANSI/MIL-STD 1815A Ada*, 1986.
9. E. Astesiano and G. Reggio, "SMoLCS-Driven Concurrent Calculi," *Proceedings of TAPSOFT'87, Lecture Notes in Computer Science* **249,** 169–201 (1987).
10. R. Milner, *A Calculus of Communicating Systems, Lecture Notes in Computer Science* **92** (1980).
11. G. Plotkin, "A Structural Approach to Operational Semantics," *Technical Report DAIMI-FN-19*, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.
12. E. Astesiano and G. Reggio, "The SMoLCS Approach to the Formal Semantics of Programming Languages—A Tutorial Introduction," to appear in *Proceedings of the CRAI Spring International Conference: Innovative Software Factories and Ada, Lecture Notes in Computer Science* (1986).
13. E. Astesiano and G. Reggio, "An Outline of the SMoLCS Methodology," to appear in *Proceedings of the Advanced School on Mathematical Models for Parallelism*, Rome, 1986, *Lecture Notes in Computer Science* (1987).
14. E. Astesiano, G. F. Mascari, G. Reggio, and M. Wirsing, "On the Parameterized Algebraic Specification of Concurrent Systems," *Proceedings of CAAP '85—TAPSOFT Conference, Lecture Notes in Computer Science* **185,** 342–358 (1985).
15. E. Astesiano, G. Reggio, and M. Wirsing, "Relational Specifications and Observational Semantics," *Proceedings of MFCS'86, Lecture Notes in Computer Science* **233,** 209–217 (1986).
16. R. Milner, "Calculi for Synchrony and Asynchrony," *Theor. Comp. Sci.* **25,** 267–310 (1983).
17. F. Morando, "An Interpreter for Concurrent Systems SMoLCS Specifications," master's thesis (in Italian), University of Genoa, Italy, 1986.
18. H. Hussmann, "Rapid Prototyping for Algebraic Specifications RAP System User's Manual," *Internal Report MIP 8502*, Universität Passau, Federal Republic of Germany, 1985.
19. M. Broy and M. Wirsing, "On the Algebraic Specification of Finitary Infinite Communicating Sequential Processes," *Proceedings of the IFIP TC2 Working Conference on "Formal Description of Programming Concepts II"* (D. Bjørner, Ed.), North-Holland Publishing Co., Amsterdam, 1983.
20. M. R. Burstall and J. A. Goguen, "The Semantics of Clear, a Specification Language," *Proceedings of Advanced Course on Abstract Software Specifications*, Copenhagen, *Lecture Notes in Computer Science* **86,** 292–332 (1979).
21. M. Broy and M. Wirsing, "Algebraic Definition of a Functional Programming Language and Its Semantic Models," *R.A.I.R.O. (Revue Française d'Automatique d'Informatique et de Recherche Operational)* **17,** 137–161 (1983).
22. M. Broy and M. Wirsing, "Partial Abstract Types," *Acta Informat.* **18,** 47–64 (1982).
23. E. Astesiano and G. Reggio, "Comparing Direct and Continuation Styles for Concurrent Languages," *Proceedings of STACS '87, Lecture Notes in Computer Science* **247,** 311–322 (1987).

Received February 25, 1987; accepted for publication May 20, 1987

Egidio Astesiano *University of Genoa, Department of Mathematics, Via L. B. Alberti 4, 16132 Genoa, Italy.*
Professor Astesiano is currently Professor of Computer Science in the Department of Mathematics of the University of Genoa. He received a laurea degree cum laude in mathematics at the University of Genoa with a thesis on partial differential equations, a field in which he spent the first two years of his career. In 1971 he switched his interests to computer science and spent two years, from 1973 to 1975, as a Visiting Fellow in the Department of Computer Science at the University of Warwick, United Kingdom. Returning to the University of Genoa in 1975, he began teaching computer science and established a research group in theoretical computer science, an activity which led in 1985 to a degree in computer science as a first step toward the formation of a Department of Computer Science. In 1985 Professor Astesiano was elected a permanent member of the IFIP WG 2.2 on Formalization of Programming Concepts. Since 1982 he has led a project on a new methodology for the specification of concurrent systems and languages which has been applied, under his leadership, in the EEC project on the draft formal definition of Ada.

Gianna Reggio *University of Genoa, Department of Mathematics, Via L. B. Alberti 4, 16132 Genoa, Italy.* Mrs. Reggio is currently a graduate student candidate for a Ph.D. degree in the Department of Computer Science of the University of Pisa and is carrying out her research in the Department of Mathematics at the University of Genoa, where she studied from 1976 to 1981 and received a laurea degree cum laude in mathematics. In 1981 she joined Professor Astesiano's group and has since then worked on the mathematical models of concurrency and its specification.
Mrs. Reggio has coauthored with Professor Astesiano the SMoLCS methodology and cooperated in its application to the EEC project on the formal definition of Ada.