SMoLCS-DRIVEN CONCURRENT CALCULI

Egidio Astesiano - Gianna Reggio
Department of Mathematics, University of Genova
Via L.B. Alberti 4, 16132 Genova, Italy

Abstract It is shown how to derive, following the principles of the SMoLCS methodology, a family of calculi, suitable for the specification of concurrent systems and languages. A calculus consists basically of a language for expressing behaviours and their parallel composition together with the rewriting rules defining their semantics; formally it is a calculus associated to an algebraic parameterized specification: for every choice of the parameters we fix one calculus in the family. The distinguishing feature of our calculi is that the combinators for behaviours include functional abstraction and application, so that behaviours can be passed as arguments and obtained as results of functions; in general behaviours can be seen just as a data type and in this sense our calculi can be higher order calculi with behaviours as first class objects.

0 INTRODUCTION

0.1 *Generalities on the SMoLCS approach*

SMoLCS is an integrated methodology for the specification of concurrent systems and languages developed mainly by the authors ([AR1, AR3]), in cooperation with M.Wirsing ([AMRW, ARW1]).
The typical fields of application of SMoLCS are large systems, multilevel architectures built from systems with different granularity, complex concurrent languages with modules and interference between sequential and concurrent features.
For the specification of concurrent systems, SMoLCS has been applied to specify the internode communication architecture of the project Cnet (a local net of workstations) (see [AMRZ1, AMRZ2]).
As a method for the specification of languages, it is the methodology chosen for the formal definition of the dynamic semantics of Ada® in the CEC-MAP project ([AGMRZ, CRAI-DDC]).
The roots of SMoLCS, both for inspiration and technical ideas, are in the work of Milner on CCS and SCCS [M1, M2], of Plotkin [P] on SOS, of Broy and Wirsing on partial data types [BW1, BW2] and of Wirsing and Sannella on algebraic specification languages [W, SW]. On these roots SMoLCS has grown into a precise coherent framework, whose distinguishing features we briefly summarize.
The specification of a system is obtained as an instantiation of a parameterized data type, following a schema based on an operational intuition of a process as a labelled transition system and of a concurrent system as resulting from the composition of the component subsystems. The abstraction from the

---

Work partly funded by CNR Italy and MPI 40%.

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

operational intuition is obtained by a schema ensuring the existence of an observational semantics, represented by an algebra.

By concurrent system we mean a labelled transition system built from some subcomponents; these subcomponents are of two kinds: active, called processes, and passive, called global objects. Each active subcomponent is in turn modelled as a labelled transition system. A transition represents an action and the difference between the two kinds of subcomponents is that the passive ones cannot perform any transition by themselves; they change their states only as a consequence of a process transition.

A state of a concurrent system is modelled as a set of states corresponding to its subcomponents; the transitions are inferred from the transitions of the active subcomponents in three steps: synchronization, parallelism, monitoring. This SMoLCS schema can be expressed in an algebraic parameterized way so that every instantiation on the appropriate parameters, defining the information for synchronization, parallelism and monitoring, is an abstract data type (see [AMRW, ARW3]).

The definition of a SMoLCS specification of a system is modular and hierarchical. More precisely every composition step is a parameterized abstract data type specification: for example the synchronization step STS takes as parameters the specification of a transition system PROC-SYST (representing the processes) and an algebraic specification GOBJ (representing the global objects) and gives a labelled transition system STS(PROC-SYST,GOBJ) whose transitions represent the synchronous interactions between processes.

Together with an initial algebra semantics, corresponding to an operational semantics, the SMoLCS approach supports, with explicit linguistic constructs, the definition of an observational semantics again via a parameterized abstract data type specification, where the parameters correspond to a formalization of the observations. Every instantiation of such schema admits a terminal model, called concurrent algebra, in which two states of the concurrent system are equivalent if and only if they satisfy the same observations; moreover every subcomponent of the state gets an observational semantics by closure with respect to state contexts (see [ARW1] for foundations). Note that this is just an existential definition, to guarantee consistency; for any instantiation such observational semantics has to be characterized more explicitly, by suitable equivalences on the derivation trees associated to states and subcomponents. The above schema permits to formalize observationally various semantics as input/output, streams semantics, strong equivalence, classes of bisimulation equivalences and test semantics.

The SMoLCS semantic definition of a language is compositional (i.e. is a homomorphism from a syntax algebra into a semantic algebra). It is done in two steps (see [AR1]): in the first a set of clauses, called denotational clauses, one for each syntactic clause, defines a translation into an intermediate language, with appropriate combinators for handling concurrency; in the second, a SMoLCS specification is given of the abstract concurrent system corresponding to program executions, with its semantics defined by an appropriate concurrent algebra. The denotational clauses can be given both in the Oxford continuation style ([AR1]) and in the VDM-like direct semantic style ([AR2]) also for a comparison; but note that they can be seen just as algebraic axioms, that the general semantic definition is just the specification of an abstract data type.

0.2 *Concurrent calculi*

The main aim of SMoLCS is to provide a precise formal framework which can be adapted to the level of the system to be specified. Then, since the overall approach is that of partial abstract data types [BW2], in order to derive the properties of the specified objects we can use an adaptation of the usual machinery of partial abstract data type specifications. This machinery consists mainly of the proof techniques associated to a specification seen as a logical rewriting system and of the associated tools.

In particular for the specifications we use in SMoLCS, an initial model always exists where equality and definedness coincide with provable equality and definedness. Hence a calculus is naturally associated to each of our specifications; because of the form of the axioms, it can be seen that this calculus corresponds to an operational semantics for the specified concurrent system.

As for the tools a specific rapid prototyping tool has been developed for SMoLCS ([Mo]) which is a variation of the RAP system [H], specially tailored to the structure of SMoLCS. It consists of a concurrent symbolic intepreter, which can derive transitions for a specified concurrent system, and of a translator which, taken the denotational clauses specifying the semantics of a language, can convert a source program into a program written in the intermediate concurrent language.

In the above sense a SMoLCS calculus is the one associated to a SMoLCS specification. In the first part of the paper we illustrate this point of view by means of an example, which also introduces the use of processes as data types and of functional combinators.

Correspondingly to the parameterization principle of SMoLCS, within a schema for defining synchronization, parallelism and monitoring, one can define for each specification appropriate combinators on processes. This possibility enhances flexibility and allows to write high level specifications, without the need of, so to speak, translating into a fixed language. However all this freedom has its own disadvantages, especially for deriving properties of the specified system. Indeed, to this end it is much easier to have a fixed set of combinators, with a well established set of properties. We propose in this paper a balance between these two attitudes, consisting in a family of calculi, where we have a fixed set of combinators for describing a kind of basic processes, called behaviours; but where there is room for fixing some parameters related to various data structures and to decisions about the interactions between processes. The result is a parameterized calculus, which is introduced in the second part of the paper. The two essential features of this calculus are the use of functional combinators and the possibility of having processes as data types. One of these calculi has been used as the intermediate language in the two steps SMoLCS definition of Ada [CRAI-DDC], a project where we have learnt, for example, that the use of functional combinators is essential for giving denotations to procedures as functions from values into processes, and for keeping high-level and modular the definition

Then in the third part we begin to study the properties of our combinators w.r.t. a basic observational equivalence, corresponding to the strong bisimulation equivalence of Milner and Park. Under some restrictions, various properties of combinators are shown, nicely corresponding to our intuition. But all the given properties hold without restrictions for the full calculi and a generalized notion of strong equivalence; the complete theory will appear in a more technical paper.

1 SMoLCS SPECIFICATIONS AND CALCULI

Technical preliminaries. In the following we refer to [BW2] for a precise definition of the concepts related to abstract data type techniques; but let us give some informal explanations. By an abstract data type specification we mean a signature and a set of axioms. Since we use a partial data type approach, i.e. the value of an operation can be undefined for some arguments, also definedness predicate symbols are used, one for each sort, to say that an object is defined; all are indicated by **D** (the sorts can be deduced from the context). Axioms are always first order formulas in positive conditional form, i.e. of the form $\wedge_i e_i \supset e$, where e can have form either $\mathbf{D}(t)$ or $t_1 = t_2$ and e_i can have form either $\mathbf{D}(t_i)$ or $(\mathbf{D}(t_i) \wedge t_i = t_i')$. It is assumed that every axiom is implicitly universally quantified over all variables, but variables can only range over defined values in the interpretation. Terms and axioms are interpreted in partial algebras, which are structures consisting of a set of carriers, one for each sort in the signature, and a set of (partial) functions corresponding to the interpretations of the operation symbols (including the definedness predicates, which are assumed to be total, i.e. either true or false on every element). There are two other important points about interpretation: first, $t_1 = t_2$ is true iff either both t_1 and t_2 are defined and equal or both are undefined; second, the functions are strict, i.e. if $Op(t_1,...,t_n)$ is defined, then all $t_1,...,t_n$ are defined.

In the following we will use some notations which we now explain.

Let S, O, F be respectively a set of sorts, of operations and of positive conditional axioms and $A = (\Sigma_A, F_A)$, $B = (\Sigma_B, F_B)$ be two specifications: then

- **sorts** S **opns** O **axioms** F denotes the specification having for signature (S,O) and axioms F;
- A+B denotes the specification having for signature $(Sorts(\Sigma_A) \cup Sorts(\Sigma_B), Opns(\Sigma_A) \cup Opns(\Sigma_B))$ and for axioms $F_A \cup F_B$;
- **enrich** A **by sorts** S **opns** O **axioms** F denotes the specification

  A + (**sorts** $S \cup Sorts(\Sigma_A)$ **opns** $O \cup Opns(\Sigma_A)$ **axioms** F);
- A[srt'/srt] denotes the specification A where the sort srt is renamed srt'.

A running example. By means of a concrete example we first illustrate the main features of a SMoLCS specification, i.e. of a specification of a concurrent system obtained by instantiating the SMoLCS parameterized schema.

The example is also meant to show how processes can be considered as data and later will be extended to handle parameterized process types, by introducing an algebraic version of function spaces.

Finally we will discuss how a concurrent calculus is associated to the given specification.

As an example we consider the formal description of a class of simple concurrent architectures, indicated by PD. Each architecture consists of (a varying number of) processes and (a fixed number) of buffers shared among processes. Each process has a local (private) memory and an instruction part defining its activity. Processes can communicate between them by exchanging messages in a synchronized way through channels (handshaking communication) and by writing and reading the buffers. The exchanged messages and the buffer contents are just values; values are natural numbers and also the processes

themselves. Processes execute their instructions in a completely free parallel way, except when they try to communicate between them or to get access to the buffers (several contemporaneous accesses to the same buffer are not allowed).
First we define processes and then show how to compose them into concurrent systems representing PD architectures.

1.1 *Processes*

A set of processes is described by an algebraic (labelled) transition system. An algebraic transition system is an algebraic specification with two sorts, state (the states of the system) and flag (the labels), and a boolean operation $\square \xrightarrow{\square} \square$: state × flag × state → bool defining the system transitions. In the following the transitions will be defined by sets of axiom schemata of the form

$$\text{" cond} \supset s \xrightarrow{f} s' = \text{true "}$$

to be interpreted: if the condition cond (a conjunction of equations) is true, then the transition $s \xrightarrow{f} s'$ belongs to the system. Notation: $s \xrightarrow{f} s' = \text{true}$ is usually written $s \xrightarrow{f} s'$.
It is important to note that a transition has the following intuitive meaning: a process in a state s has the capability of moving to a state s' by an action whose interaction with the external environment is represented by the flag f; hence f is conveying both information on the conditions of the environment which allow the capability to become effective and on the transformation of the environment produced by the execution of that action. This meaning of labelled transitions has now become classical after its use in CCS [M1, M2] and in SOS [P]; we will now illustrate it by few examples. A capability of reading the content of a shared buffer by a process pr can be written as

$$pr \xrightarrow{READBUF(b,v)} pr'$$

where b is a buffer identifier and v a value.
Note that, as it will be defined later in the synchronization step, this capability will become effective only in an external environment where the content of the buffer b is exactly the value v.
A capability of writing on a shared buffer by a process pr can be written as

$$pr \xrightarrow{WRITEBUF(b,v)} pr'$$

where b is a buffer identifier and v a value.
Analogously we can express the well known capabilities of handshake communication

$$pr \xrightarrow{SEND(c,v)} pr' \text{ (sending)} \quad \text{and} \quad pr \xrightarrow{REC(c,v)} pr' \text{ (receiving)}$$

where c is a channel identifier and v a value.
An example of conditional rule is

$$pr_1 \xrightarrow{f} pr_1' \supset pr_1 ; pr_2 \xrightarrow{f} pr_1' ; pr_2$$

which defines, inductively, the capabilities of a process executing the concatenation of two statements.
Example. As an example of algebraic transition system we give PD-PROC, which describes the PD processes, mentioned before.
The states of PD-PROC are defined by the following algebraic specification, which is an example of recursive specification. The use of recursive algebraic specifications is quite natural, when trying to give specifications in a modular way. For example in the following specification we first say that processes are

couples of instructions and local memories; after that we define local memories as finite maps from locations into values and then we say that values consists of processes as data and of natural numbers. Since we use a recursive definition, we expect that the resulting specification (signature and axioms) is in some sense, a well determinated fixpoint of the transformation associated to the definition. It is not difficult to see that, under some natural conditions, such fixpoint exists. For a more technical discussion we refer to the Appendix 1, where also a non recursive version (i.e. the explicit fixpoint) of the following specification is given.

As it should be clear from the intuitive explanations, given two specifications $ELEM_1$, $ELEM_2$ with main sorts $elem_1$ and $elem_2$, $PROD(ELEM_1,ELEM_2)$ indicates the parametric specification of the cartesian products with main sort $prod(elem_1,elem_2)$ and operations $<\Box,\Box>$ (pair constructor) and Sel_1, Sel_2 (component selectors); $MAP(ELEM_1,ELEM_2)$ indicates the parametric specification of finite maps with main sort $map(elem_1,elem_2)$ and operations $\Box[\Box/\Box]$ (substitution) and $\Box(\Box)$ (application). LOC (locations), BUFID and CHID (buffer and channel identifiers) are specifications which are not further defined here. The main sort of a specification is just a sort of the specification, used in defining parametric specifications. The use of the $\Box$'s indicates that some operations have an infix syntax; "all total" stands for the set of axioms having form $\mathbf{D}(Opn(x_1,...,x_n))$ requiring the totality of all operations appearing in the **opns** part.

PROC = **enrich** PROD(INSTR,LMEM)[proc/prod(instr,lmem)] **by**
- **opns** Nil: $\rightarrow$ proc
- **axioms** $\mathbf{D}$(Nil)

LMEM = MAP(LOC,VALUE)[lmem/map(loc,value)]

VALUE = **enrich** PROC + NAT **by**
- **sorts** value
- **opns** Pval: proc $\rightarrow$ value
  - Nval: nat $\rightarrow$ value
  - $\{Op: value \times ... \times value \rightarrow value \mid Op: nat \times ... \times nat \rightarrow nat \in Sig(NAT)\}$
- **axioms** $\mathbf{D}(Pval(pr))$ $\mathbf{D}(Nval(n))$
  - $\{Op(Nval(n_1),...,Nval(n_k)) = Nval(Op(n_1,...,n_k)) \mid Op: nat \times ... \times nat \rightarrow nat \in Sig(NAT)\}$

INSTR = **enrich** LOC + VALUE + BUFID + CHID **by**
- **sorts** instr
- **opns**
  1. Writebuf, Readbuf: loc $\times$ bufid $\rightarrow$ instr
  2. Send, Rec: loc $\times$ chid $\rightarrow$ instr
  3. Skip: $\rightarrow$ instr
  4. Start: proc $\rightarrow$ instr
  5. $\Box\,;\,\Box$, $\Box + \Box$: instr $\times$ instr $\rightarrow$ instr
  6. While $\Box \neq 0$ Do $\Box$: loc $\times$ instr $\rightarrow$ instr
  7. $Seq\text{-}Instr_1$: ... $\rightarrow$ instr

  ...

  7+n $Seq\text{-}Instr_n$: ... $\rightarrow$ instr
- **axioms** "all total"
  - $i_1 ; (i_2 ; i_3) = (i_1 ; i_2) ; i_3$ Skip ; i = i
  - $i_1 + i_2 = i_2 + i_1$ $i_1+(i_2+i_3) = (i_1+i_2)+i_3$.

Comments.

1. Writing the content of a cell of the local memory in a buffer and reading the content of a buffer together with storing it in a cell of the local memory (buffers contain values).
2. Sending and receiving messages (just values) through channels.
3. Skip is the usual null instruction.
4. Creation of a new process.
5. Sequential composition and nondeterministic choice.
6. While the content of the location is different from 0 execute the given instrucion.

7,..,7+n. Sequential instruction, i.e. instructions whose executions do not require either interactions with other processes or with the buffers.

Note that the processes can store into the buffers or into the local memory and exchange between them other processes because a value can be a process. End of comments.

Now we give the specification of the transition system of PD processes, indicated by PD-PROC. The flags of PD-PROC (specification PFLAG) are in correspondence with the executions of the concurrent instructions, exception made for TAU (Milner's internal action) which corresponds to the execution of sequential instructions. For simplicity we do not report the full specification PFLAG; it can be easily understood by looking at the axioms of PD-PROC.

PD-PROC =
enrich PROC + PFLAG **by**

opns $\square \xrightarrow{\square} \square : proc \times pflag \times proc \rightarrow bool$

axioms

1 $\langle Writebuf(l,b), lm\rangle \xrightarrow{WRITEBUF(lm(l),b)} \langle Skip, lm\rangle$

2 $\langle Readbuf(l,b), lm\rangle \xrightarrow{READBUF(v,b)} \langle Skip, lm[v/l]\rangle$

3 $\langle Send(l,c), lm\rangle \xrightarrow{SEND(lm(l),c)} \langle Skip, lm\rangle$

4 $\langle Rec(l,c), lm\rangle \xrightarrow{REC(v,c)} \langle Skip, lm[v/l]\rangle$

5 $\langle Start(pr), lm\rangle \xrightarrow{START(pr)} \langle Skip, lm\rangle$

6 $Nil \xrightarrow{CREATED(pr)} pr$

7 $\langle i_1, lm\rangle \xrightarrow{pf} \langle i_1', lm'\rangle \supset \langle i_1 ; i, lm\rangle \xrightarrow{pf} \langle i_1' ; i, lm'\rangle$

8 $\langle i_1, lm\rangle \xrightarrow{pf} \langle i_1', lm'\rangle \supset \langle i_1 + i, lm\rangle \xrightarrow{pf} \langle i_1', lm'\rangle$

9 $Iszero(lm(l)) = true \supset \langle While\ l \neq 0\ Do\ i, lm\rangle \xrightarrow{TAU} \langle Skip, lm\rangle$

10 $Iszero(lm(l)) = false \supset \langle While\ l \neq 0\ Do\ i, lm\rangle \xrightarrow{TAU} \langle i ; While\ l \neq 0\ Do\ i, lm\rangle$

$\{ \langle Seq\text{-}Instr_j(...), lm\rangle \xrightarrow{TAU} \langle i_j', lm_j'\rangle \mid 1 \leq j \leq n \}.$

Comments. A process in a state <i,lm>, where i is an input instruction (Readbuf,Rec), can perform nondeterministically an action out of an infinite set, one for every possible value which can be received (axioms 2,4).

Axiom 5 defines the capability of a process of creating a new process. Axiom 6 defines the capability of being created, which is represented by a transition of the process Nil into the initial state of the created process and is denoted by the flag CREATED(...).

Axioms 7 and 8 completely define the sequential composition and the nondeterministic choice, because of Skip is the identity of **;** and + is commutative (look at the axioms of INSTR).

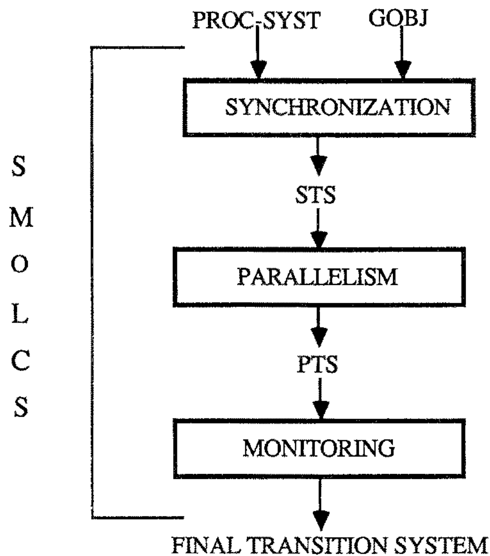
Axioms 9 and 10 define the While instruction. End of comments.

1.2 *Concurrent systems*

Now that we have defined processes, we show how to compose them into concurrent systems, of which processes are subcomponents. In a concurrent system a state consists of the states of the processes plus the state of the global object; we choose to represent it as a pair $<\{pr_1,pr_2,...,pr_n\},go>$ where go is the global object and $\{pr_1,pr_2,...,pr_n\}$ is a multiset of states of processes.

Now, assuming that a transition system PROC-SYST, representing the subcomponent processes and an algebraic specification GOBJ, representing the global object, are given, how do we specify the resulting composed system?

Our idea is to split the composition in some steps. First the actions of the processes are composed producing new actions; this step is conveniently subdivided into two other steps: one (synchronization) defines the actions resulting from some synchronized cooperation between processes; another (parallel composition) defines which are the synchronous actions that can happen in parallel. Then a third step (monitoring) defines which actions resulting after the second step are allowed to happen as actions of the whole system.



Example. As an example of concurrent system we report the definition of PD. Following the above schema the definition is split in four parts which are reported and commented in sections: 1.1 (the algebraic transition system given before and corresponding to the process subcomponents), 1.2.1 (synchronization), 1.2.2 (parallelism), 1.2.3 (monitoring).

1.2.1 *Synchronization*

We define the synchronous actions by giving a new transition system STS, where the transition relation $\longrightarrow$ corresponds to synchronous actions, starting from a transition system PROC-SYST (representing the

component processes with transition relation $\rightarrow$) and an algebraic specification GOBJ (representing the global object). The states of STS are pairs <prms,go> where ms is a multiset of proces states and go a state of the global object.

The synchronous actions are given by the a set of axioms having form

$$(\bigwedge_{1\leq j\leq n} pr_j \xrightarrow{f_j} pr_j') \wedge cond(sf,\{f_1,...,f_n\},go) \supset <\{pr_1,...,pr_n\},go> \xrightarrow{sf} <\{pr_1',...,pr_n'\},go'>$$

where $cond(sf,\{f_1,...,f_n\},go)$ is a conjunction of equations and represents the condition under which the process actions $pr_1 \xrightarrow{f_1} pr_1',...,pr_n \xrightarrow{f_n} pr_n'$ can synchronize.

Note that the transformation of the global object associated to a synchronous action can be nondeterministic, i.e. it is possible to have also another axiom similar to the above one except that go' is replaced by a different go" .

We briefly illustrate the idea referring to the example capabilities of subsection 1.1.

The effect of reading a shared buffer can be defined by:

$$pr_1 \xrightarrow{READBUF(b,v)} pr_1' \wedge go(b) = v \quad \supset <\{pr_1\},go> \xrightarrow{READBUF(b,v)} <\{pr_1'\},go>$$

where in the global object are recorded the states of the shared buffers; note that the flag of the resulting action is still READBUF(b,v) (thus recording the kind of the action) since the effective happening of the action will still depend on the actions of the other processes because two contemporaneous readings of the same buffer are mutually exclusive; we will handle that in the parallel composition step.

The effect of writing on a shared buffer is defined by:

$$pr_1 \xrightarrow{WRITEBUF(b,v)} pr_1' \supset <\{pr_1\},go> \xrightarrow{WRITEBUF(b,v)} <\{pr_1'\},go[v/b]>,$$

go[v/b] represents the state of the global object where the content of the buffer b has been changed in v.

The effect of a handshaking communication is defined by

$$pr_1 \xrightarrow{SEND(c,v)} pr_1' \wedge pr_2 \xrightarrow{REC(c,v)} pr_2' \supset <\{pr_1,pr_2\},go> \xrightarrow{TAU} <\{pr_1',pr_2'\},go>$$

where the synchronous flag TAU reminds Milner's symbol for an internal action, i.e. an action with no interaction with the external environment. Moreover no other synchronous actions involving SEND(c,v) or REC(c,v) are defined, thus a SEND(c,v) action of a behaviour can be executed only together with a REC(c,v) action of another behaviour.

Note that also the process internal actions will become synchronous actions, as defined below

$$pr_1 \xrightarrow{TAU} pr_1' \supset <\{pr_1\},go> \xrightarrow{TAU} <\{pr_1'\},go>.$$

Creation and termination of component processes are handled by defining a particular process state Nil with the property $<\{Nil,pr_1,...,pr_n\},go> = <\{pr_1,...,pr_n\},go>$, and synchronous actions such as

$$Nil \xrightarrow{CREATED(pr)} pr \wedge pr_1 \xrightarrow{START(pr)} pr_1' \supset <\{Nil,pr_1\},go> \xrightarrow{TAU} <\{pr,pr_1'\},go>.$$

Example. The synchronous interactions between the processes of PD are described by the following algebraic transition system PD-STS.

The states of PD-STS are defined by the specification PD-STATE and its transitions are labelled by elements of sort pflag of the specification PFLAG (introduced in subsection 1.1).

PD-STATE = **enrich** PROD(MSET(PROC),BUFFERS)[state/prod(mset(proc),buffers)] **by**
axioms $<\{Nil\}\cup bhms,bfs> = <bhms,bfs>$.

The global object records the contents of the shared buffers.

MSET(ARG) indicates the parametric specification of multisets with operations $\{\square\}$ (singleton multiset constructor) and $\square \cup \square$ (union). Notation $\{a_1\} \cup ... \cup \{a_n\}$ is simply written $\{a_1, ..., a_n\}$.

BUFFERS = MAP(BUFID,VALUE)[buffers/map(bufid,value)]

PD-STS = **enrich** PD-STATE + PD-PROC **by**

opns $\square{=}\square{\Rightarrow}\square$: state × pflag × state → bool

axioms

$pr \xrightarrow{TAU} pr' \supset \langle\{pr\},bfs\rangle =TAU\Rightarrow \langle\{pr'\},bfs\rangle$

$pr \xrightarrow{READBUF(v,b)} pr' \wedge bfs(b) \equiv v \supset \langle\{pr\},bfs\rangle =READBUF(v,b)\Rightarrow \langle\{pr'\},bfs\rangle$

$pr \xrightarrow{WRITEBUF(v,b)} pr' \supset \langle\{pr\},bfs\rangle =WRITEBUF(v,b)\Rightarrow \langle\{pr'\},bfs[v/b]\rangle$

$pr_1 \xrightarrow{SEND(v,c)} pr_1' \wedge pr_2 \xrightarrow{REC(v,c)} pr_2' \supset \langle\{pr_1,pr_2\},bfs\rangle =TAU\Rightarrow \langle\{pr_1',pr_2'\},bfs\rangle$

$pr_1 \xrightarrow{START(pr)} pr_1' \wedge Nil \xrightarrow{CREATED(pr)} pr \supset \langle\{pr_1,Nil\},bfs\rangle =TAU\Rightarrow \langle\{pr_1',pr\},bfs\rangle$.

1.2.2 *Parallelism*

Intuitively by means of this composition operation we define whether two actions can be executed in parallel (without synchronization). The actions to be considered for composition are, inductively, the actions of the synchronized system STS and the new actions already obtained by parallel composition.

As before for synchronization we can describe the operation of parallel composition as producing a new system PTS from the system STS. PTS is simply given by augmenting the transitions of STS (indicated by $=>$) with the new elements, which are given by a set of axioms having the following form

$$\langle prms_1,go\rangle -sf_1\rightarrow \langle prms_1',go_1'\rangle \wedge \langle prms_2,go\rangle -sf_2\rightarrow \langle prms_2',go_2'\rangle \supset$$
$$\langle prms_1 \cup prms_2,go\rangle -sf_1//sf_2\rightarrow \langle prms_1' \cup prms_2',go'\rangle$$

provided we have given the partial (binary, commutative and associative) operation // on the flags of the synchronous actions.

In our previous examples a writing action on a shared buffer and a handshaking communication can be executed together giving a new composed action, which in turn can be executed together with an handshaking communication of some other processes. On the converse a reading or updating action of a shared buffer does exclude whatever other access of the same buffer.

Example. The allowed contemporaneous executions of the synchronized interactions between the processes of PD are described by the algebraic transition system PD-PTS.

The states of PD-PTS are the same of PD-STS; the transition relation and the flags of PD-PTS are an enrichment of those of PD-STS.

The transformation of the buffers associated to a parallel action in the system PD-PTS corresponds to execute the transformations associated to the component synchronous actions in some order; note that the result does not depend on the chosen order.

In the specification PD-TS Eq indicates an explicit total equality operation of the specification BUFID with functionality bufid × bufid → bool.

PD-PTS = **enrich** PD-STS **by**

opns $\square // \square$: pflag $\times$ pflag $\rightarrow$ pflag

Isacc: pflag $\times$ bufid $\rightarrow$ bool

axioms

$f_1//f_2 = f_2//f_1 \quad (f_1//f_2)//f_3 = f_1//(f_2//f_3)$

$Isacc(TAU,b) = false$

$Isacc(READBUF(v,b_1),b_2) = Eq(b_1,b_2)$

$Isacc(WRITEBUF(v,b_1),b_2) = Eq(b_1,b_2)$

$Isacc(f_1//f_2,b) = Isacc(f_1,b) \vee Isacc(f_2,b)$

$\langle mpr_1,bfs\rangle =TAU\Rightarrow \langle mpr_1',bfs\rangle \wedge \langle mpr_2,bfs\rangle =pf\Rightarrow \langle mpr_2',bfs'\rangle \supset$

$\langle mpr_1 \cup mpr_2,bfs\rangle =pf//TAU\Rightarrow \langle mpr_1' \cup mpr_2',bfs'\rangle$

$\langle mpr_1,bfs\rangle =READBUF(v,b)\Rightarrow \langle mpr_1',bfs\rangle \wedge$

$\langle mpr_2,bfs\rangle =pf\Rightarrow \langle mpr_2',bfs'\rangle \wedge Isacc(b,pf) = false \supset$

$\langle mpr_1 \cup mpr_2,bfs\rangle =READBUF(v,b)//pf\Rightarrow \langle mpr_1' \cup mpr_2',bfs'\rangle$

$\langle mpr_1,bfs\rangle =WRITEBUF(v,b)\Rightarrow \langle mpr_1',bfs[v/b]\rangle \wedge$

$\langle mpr_2,bfs\rangle =pf\Rightarrow \langle mpr_2',bfs'\rangle \wedge Isacc(b,pf) = false \supset$

$\langle mpr_1 \cup mpr_2,bfs\rangle =WRITEBUF(v,b)//pf\Rightarrow \langle mpr_1' \cup mpr_2',bfs'[v/b]\rangle$.

1.2.3 *Monitoring*

Here we take into consideration any form of global control, by which only some of the actions which are locally possible in a system (i.e. those obtained by (synchronization and) parallel composition) are allowed to become actions of the overall system. It is at this step that we can, for example, define an interleaving mode, admitting only one synchronized action at time, or a mode in which all actions that can be executed together do so. Here we can also define that the buffer reading actions take precedence over the buffer writing actions (i.e. when in a state it is possible a reading action on a buffer, a writing action on the same buffer will never be allowed).

As before we can define the monitoring operation by giving a new transition system MTS (with transition relation $===\rangle$) starting from a parallel system PTS (with transition relation $=\Rightarrow$). The states of MTS are the same of PTS. The transitions of the new system are defined by giving some axioms following this schema:

$\langle prms_1,go\rangle -sf\rightarrow \langle prms_1',go'\rangle \wedge cond(sf,\langle prms_1 \cup prms_2,go\rangle,extf) \supset$

$\langle prms_1 \cup prms_2,go\rangle ==extf==\rangle \langle prms_1' \cup prms_2,go'\rangle$.

Note that this axiom schema specifies, as it was anticipated informally, that an action of the system is determined by an action of a part of the component processes; here the partial action is $\langle prms_1,go\rangle -sf\rightarrow \langle prms_1',go'\rangle$ and $prms_2$ is the multiset of the states that do not cooperate to that action.

Moreover the monitoring decision must depend only on the action capabilities of the processes present in a system state and not on their states.

Example. We specify a parallel mode for the execution of the processes of the PD architectures (i.e. every parallel action is allowed to become an action of the system) defining the concurrent transition system PD.

The states of PD are still defined by the specification PD-STATE and its transitions are labelled by elements of the following specification EXTFLAG.

EXTFLAG = **sorts** extflag **opns** TAU: $\to$ extflag **axioms** **D**(TAU)

PD = **enrich** PD-PTS + EXTFLAG **by**

opns $\square$ ==$\square$==>$\square$: state $\times$ extflag $\times$ state $\to$ bool

axioms <prms$_1$,bfs> ==pf==> <prms$_1$,bfs'> $\supset$ <prms$_1$ $\cup$ prms$_2$,bfs> ==TAU==> <prms$_1$$\cup$ prms$_2$,bfs'>.

1.3 *Semantics and calculi*

To the specification of PD we can first associate a semantics, given by its initial model; this model indeed exists and corresponds roughly to an operational semantics modulo the initial congruence on the states of the system.

Proposition1. (see [ARW1]) There exists an initial model I_{PD} of the specification PD such that I_{PD} is term generated and for any t, t_1, $t_2 \in W_{Sig(PD)}$

$I_{PD} \models \mathbf{D}(t)$ iff PD $\vdash \mathbf{D}(t)$ and

PD $\vdash \mathbf{D}(t_1) \wedge \mathbf{D}(t_2)$ implies ($I_{PD} \models t_1 = t_2$ iff PD $\vdash t_1 = t_2$).

In particular for any st, st' $\in W_{Sig(PD)}|_{state}$, pr, pr' $\in W_{Sig(PD)}|_{proc}$ and pf $\in W_{Sig(PD)}|_{pflag}$

PD $\vdash$ st =TAU=> st' iff $I_{PD} \models$ st =TAU=> st' and

PD $\vdash$ pr $\xrightarrow{pf}$ pr' iff $I_{PD} \models$ pr $\xrightarrow{pf}$ pr'. $\square$

The proposition shows that the specification PD defines an associated calculus, which we indicate by ***PD***, corresponding to an operational semantics, and formally consisting of the equality $=$ and of the transitions, both of processes and architectures, provable in PD. In general for any specification SYST of a system, we will call ***SYST*** the corresponding calculus. (These are the calculi to which the rapid prototyping tool ([Mo]) developed for SMoLCS specifications applies).

Assume now that we want to consider two architectures to be equivalent iff they have the same input/output relation, where the inputs and the outputs are respectively the initial numeric contents of the buffers and the lists of the intermediate numeric contents of the buffers. Then we have to define an observational semantics of PD.

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

- a specification, defining the observations on the system (here PD-PLUS), by means of boolean relations (here Res) stating that some observation values (here lists of numeric buffer contents) are true of some observed objects (here the states of PD) (see [ARW1]);

OBS = **enrich** MAP(BUFID,NAT)[obs/map(bufid,nat)] **by**

opns $\square$^$\square$: obs $\times$ obs $\to$ obs

axioms **D**(ob$_1$^ob$_2$)

PD-PLUS = **enrich** PD +OBS **by**

opns Res: state × obs → bool

Val: buffers → obs

axioms Val(Empty_Map) = Empty_Map

Val(bfs[Pval(pr)/b]) = Val($bfs|_b$)

Val(bfs[Nval(n)/b]) = Val(bfs)[Nval(n)/b]

Res(<prms,bfs>,Val(bfs)) = true

<prms,bfs>==TAU==> st ∧ Res(st,ob) = true ⊃ Res(<prms,bfs>,Val(bfs)^ob) = true.

($bfs|_b$ represents the map bfs where every association to b is dropped);

- a definition of 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;

- the definition of the observational semantics as the minimally defined and term generated algebra (here CALG) terminal in that class; a basic general theorem (in [ARW1]) shows that this algebra has indeed the properties required of an observational semantics.

Then we obtain the following result qualifying CALG as the observational semantics of PD w.r.t. the observations, expressed by the operation Res (here we have chosen the initial model of OBS).

Proposition 2.([ARW1]) There exists an algebra CALG with the following properties:

for any srt ∈ Sorts(PD-STATE-Sorts(OBS), ground terms t, t' ∈ $W_{Sig(PD\text{-}STATE)}|_{srt}$

01 CALG |= **D**(t) iff PD |– **D**(t)

02 CALG |= t = t' iff for any ob ∈ $W_{Sig(OBS)}|_{obs}$,

any st ∈ $W_{Sig(PD\text{-}STATE)}\{x\}|_{state}$ with x of sort srt

[PD-PLUS |– Res(st[t/x],ob) = true iff PD-PLUS |– Res(st[t'/x],ob) = true]. □

If Σ is a signature and srt a sort of Σ, then $W_{\Sigma}|_{srt}$ represents the set of all terms of sort srt built on Σ.

Property 01 says that all the interesting objects of PD-STATE are defined in CALG; by property 02 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 nonobserved subcomponent of a state gets an observational semantics: in PD, for example, this is true of processes.

Hence CALG |= st_1 = st_2 iff st_1 and st_2 produce the same outputs.

Correspondingly to the above observational semantics we could prove some useful identities between PD processes and architectures. However this may be in general rather unpractical, since it has to be done explicitly ad hoc for the specification PD. That is why in the second part of the paper we will develop a parameterized calculus starting from a fixed set of combinators, in order to be able to give standard identities w.r.t. a basic observational semantics which is a generalization of Milner and Park's strong equivalence.

1.4 *Multilevel concurrent systems*

In the previous sections we have defined a three steps procedure that, given a transition system, specifying some component processes, and a synchronization, a parallel and a monitoring specification, produces a new transition system, specifying a concurrent system. Clearly the procedure can be iterated;

if some subcomponent processes (said concurrent subcomponents) are themselves concurrent systems, then they can be specified by the same procedure. Consider for example a net of (workstations) nodes, such that in a node many processes can cooperate, possibly using a shared memory, while the nodes can exchange messages in a broadcasting or/and point to point mode. Then we can specify the net applying twice the SMoLCS procedure; in one application the subcomponents are the processes cooperating in a node and the resulting concurrent system specifies a node; in the other one the nodes, specified in the first level, become the new subcomponents and the resulting concurrent system specifies the net (as eg in [AMRZ1, AMRZ2]).

It can also be shown that the procedure can be applied inductively; hence it is possible to specify systems where a subcomponent process (said inductive concurrent subcomponent) has the same nature of the composed processes, as in CCS, i.e. where states and transitions of the final system are embedded into the states and the transitions of the system describing the processes.

1.5 *A specification with parametric process types*

We describe now an extension of PD, called PPD, obtained by enriching the values handled by PD processes, which already include the processes themselves, with parametric process types. Thus the PPD processes can exchange between them and store in the buffers and in their local memories values corresponding to parametric process types. The purpose of PPD is to introduce, on the top of the already given PD example, the use of functional combinators which will play a relevant role in the rest of the paper.

The algebraic specification of functions. A parametric process type is just a function from some parameters into processes. In our algebraic setting it is convenient and feasible to give an algebraic specification of functions from elements of some sorts into the elements of some other sort; in [ARW2] (but see also [BW3]) we study the problem and present several solutions; here we briefly introduce some basic concepts and notations.

Let ARG and RES be two algebraic specifications with main sorts arg and res respectively; by FUNCT(ARG,RES) we indicate the algebraic specification of functions from elements of sort arg into elements of sort res. The specification is nothing but an algebraic formalization of the usual rewriting rules of functional calculus with abstraction and application. The only tricky point is the following: in a term like λ x . x+5, the first occurrence of x is the first argument of λ and is only a symbol, while in x+5 x stands for a value of sort nat; thus we say that the first occurrence of it is an object of sort nat -var and we provide a merge operation Nat_Var for considering a variable symbol as an object of type nat.

FUNCT(ARG,RES) has sorts		
-	funct(arg,res)	(the functions)
-	arg-var	(the "variables of type arg")
-	arg-res-fid	(identifiers for functions from arg into res)
-	bool	(the boolean values)

and operations

- $\lambda\square.\ \square$: arg-var × res $\rightarrow$ funct(arg,res) (λ abstraction)

- Arg-Var: arg-var $\rightarrow$ arg
  (this operation embeds the "variables of type arg" into the elements of sort arg)
- $\square(\square)$: funct(arg,res) $\times$ arg $\rightarrow$ res (application)
- **if** $\square$ **then** $\square$**else** $\square$: bool $\times$ res $\times$ res $\rightarrow$ res (conditional)
- Arg-Res-Fid: arg-res-fid $\rightarrow$ funct(arg,res)
  (this operation embeds the identifiers into the elements of sort funct(arg,res))
- **rec** $\square$**.** $\square$: arg-res-fid $\times$ funct(arg,res) $\rightarrow$ funct(arg,res) (recursive function constructor).

Notation: the terms having form Arg-Var(x), Arg-Res-Fid(y), where x is a term of sort arg-var and y a term of sort arg-var-fid, are simply written x, y.

The above framework allows to write the elements of sort funct(arg,res) following the usual λ-notation; moreover the elements of sort funct(arg,res) have the basic usual properties of a functional calculus, eg α-rule and β-rule. For example

$$\lambda x \,.\, x+3 = \lambda y \,.\, y+3 \qquad (\lambda x \,.\, x+3)(2) = 5 .$$

Note that all the operations in a partial specifications are strict, i.e. $\mathbf{D}(Op(t_1,...,t_n)) \supset \mathbf{D}(t_1) \wedge ... \wedge \mathbf{D}(t_n)$ and hence also the **if** $\square$ **then** $\square$**else** $\square$ operation is strict, but that does not pose problems in defining functions, because this operation is defined by

$$cond(a) = true \supset (\lambda x \,.\, \textbf{if } cond(x) \textbf{ then } r(x) \textbf{ else } r'(x))(a) = r(a)$$

$$cond(a) = false \supset (\lambda x \,.\, \textbf{if } cond(x) \textbf{ then } r(x) \textbf{ else } r'(x))(a) = r'(a).$$

For example, consider $f = \lambda x \,.\, \textbf{if } x>10 \textbf{ then } x\text{-}10 \textbf{ else } x$, which is a term of sort funct(nat,nat) defined in FUNCT(NAT,NAT); then with the usual meaning of - , the value of 5-10 is undefined; however f(5) is a defined term of sort nat.

Functions with several parameters can also be defined using FUNCT and the parameterized specification PROD. For example, FUNCT(PROD(ARG_1,ARG_2),RES) is the specification of the functions with two arguments of sort arg_1 and arg_2 respectively into res.

The example PPD. PPD is defined in the same way of PD, exception made for the specification of values; we assume that the process type parameters are just channel and buffer identifiers and, for simplicity, that each type has only one parameter.

$VALUE^{PPD}$= **enrich** FUNCT(CHID,PROC) + FUNCT(BUFID,PROC) + NAT **by**

sorts value

opns Pval: proc $\rightarrow$ value

Nval: nat $\rightarrow$ value

{Op: value $\times$ ... $\times$ value $\rightarrow$ value | Op: nat $\times$ ... $\times$ nat $\rightarrow$ nat $\in$ Sig(NAT) }

$Ptval_1$: funct(chid,proc) $\rightarrow$ value

$Ptval_2$: funct(bufid,proc) $\rightarrow$ value

axioms "all total'

{Op(Nval(n_1),..., Nval(n_k)) = Nval(Op(n_1,...,n_k)) | Op: nat $\times$ ...$\times$ nat $\rightarrow$ nat $\in$ Sig(NAT)}.

Let us just to show two examples of the use of these process types.

The ***PPD*** calculus can be used to describe architectures where several processes perform the same computation on different values (eg an array processor architecture).

Let $pt = \lambda b \,.\, <Readbuf(l,b)\,;instr_1\,;\ldots;instr_k\,;Writebuf(l,b),Empty_Map>$, be a process type parameterized on a buffer identifier, where $instr_1\,;\ldots;instr_k$ corresponds to a complex computation on the content of the local memory location l. Then the ***PPD*** term

$<\{pt(buf_1),\ldots,pt(buf_n)\},[buf_1 \rightarrow v_1,\ldots,buf_n \rightarrow v_n]>$

describes a system where the above computation is performed in parallel on all the contents of the buffers $buf_1,\ldots,buf_n$.

Consider now the process type $pt_1 = \lambda c.\ <Rec(l,c)\,;instr_1\,;\ldots;instr_k\,;Send(l,c),Empty_Map>$ parameterized on a channel identifier; then the ***PPD*** term $<\{p_0\},[\,buf \rightarrow 0]>$, where

$p_0 = \ <Start(pt_1(c_1))\,;Send(l_1,c_1)\,;\ldots;Start(pt_1(c_n))\,;\ Send(l_n,c_n)\,;$
$\quad Rec(c_1,l_1)\,;l:=l+l_1\,;\ldots;Rec(c_n,l_1)\,;l:=l+l_1\,;Writebuf(l,buf),\ [l \rightarrow 0,l_1 \rightarrow v_n,\ldots,l_n \rightarrow v_n]>$

describes a concurrent system where the above complex computation is performed in parallel on the values $v_1,\ldots,v_n$ and the sum of the results is put into the buffer buf.

2 PARAMETERIZED CONCURRENT CALCULI

In this section we restrict the SMoLCS specifications to those where processes are built on a fixed set of combinators; the new processes are called behaviours and are still parameterized on various data structures, but we can give once for all various properties related to the given combinators.

2.1 *Behaviours and varieties of calculi*

The calculi we introduce in this section are based on the notion of behaviours (a name suggested by Milner's behaviours in CCS and in SCCS). The peculiarity of behaviours compared to the models of processes used in the examples of section 1 is that they correspond to processes without local state; they are completely determined by the atomic actions they can perform, i.e. they correspond abstractly to trees only labelled by actions. Then in our approach the processes with a local state are modelled as functions from local states into behaviours; the advantage is that we combine the level of abstraction and the expressive power of behaviours and functions together. This technique is fundamental for giving high level semantic descriptions of languages as we have shown in [AR1, AR2, CRAI-DDC]. For example, if we want to give a denotational value for a procedure, which usually involves some concurrent interaction among processes, (as it is in Ada for example), then we can model it as a function which, taken some values of the parameters, produces a behaviour. The role played by behaviours in practical applications will be illustrated by some later examples.

Instead of presenting a single calculus, we will introduce a family of calculi, which may differ fundamentally in two respects: the family is parameterized on some data structures and moreover various families of subcalculi are derivable, depending on the combinators used and on the assumptions about the parameters. For every complete choice we have a calculus corresponding to an operational semantics of a (multilevel) concurrent system, where the active subcomponents are behaviours with the corresponding peculiar properties.

We can group the parameters as follows.

- An algebraic specification DATA will represent the structure of (hence the data recorded in) the global object, the data exchanged between the behaviours, the data elaborated internally by the behaviours, the behaviour atomic actions and the interactions of the concurrent system with the external world. Formally DATA will be an algebraic specification based on BOOL (a specification of boolean values) and such that its sorts include:
  - gobj, for the states of the global object
  - act, for the atomic actions of behaviours
  - extflag, for representing the interactions with the external world.

  We assume moreover that DATA is parameterized on an algebraic specification X, which will be in every instantiation the specification of behaviours.
- Another parameter defines how the behaviour subcomponents of the system concurrently interact between them; that is described following the SMoLCS methodology as introduced in section 1. Formally this parameter, indicated by SMoLCS-SYST(PROC-SYST), is just a parametric algebraic concurrent transition system (with transition relation $\Box == \Box ==> \Box$: state × extflag × state→bool), where the parameter PROC-SYST corresponds to the algebraic transition system (with transition relation $\Box \xrightarrow{\Box} \Box$: behaviour × act × behaviour →bool) defining its active subcomponents (see subsection 1.2). It will be instantiated with the transition system defining the behaviours.
- In general the concurrent system described by the calculus is a multilevel system (see subsection 1.4); thus we need some parameter for describing the noninductive concurrent subcomponents, which are just other concurrent systems. For simplictly we consider only the cases where all the noninductive subcomponents are represented by a unique algebraic transition system (with transition relation $\Box \sim\sim \Box \sim\sim> \Box$: sstate × act × sstate→bool), indicated by SUB-SYST. Obviously SUB-SYST may be also an one-level concurrent system.

2.2 *Introducing combinators*

Here we introduce the combinators for a calculus in our family, indicated by ***SYST***, together with their informal meaning. In the following subsection, the calculus will be formally defined by an algebraic specification named SYST.

The syntax of ***SYST*** is given as the signature of a specification STATE (the states of the algebraic transition system SYST) whose sorts include state (the terms of the calculus), behaviour and gobj (active and passive subcomponents); the combinators are just the operations of this signature and in what follows we use for them the same notations used for the operations. We use **=** to indicate the provable equality in a specification.

First we give the combinators for expressing global objects and behaviours and then the combinator for composing them into a state of the concurrent system.

GLOBAL OBJECT

- All the operations of the specification DATA with functionality

  $srt_1 \times ... \times srt_n \quad \rightarrow gobj \ (n \geq 0)$

are the calculus combinators for expressing the global object.

The meaning of these combinators are given by the axioms of DATA.
In what follows SORTS indicates Sorts(DATA) $\cup$ {null}.

BEHAVIOURS

prefixing of an action

- $\Box \Delta \Box$: act $\times$ behaviour $\rightarrow$ behaviour

a Δ bh represents the behaviour which performs the action a and then behaves as specified by bh.
Behaviour atomic actions are represented by terms of sort act built on the signature of STATE.
We recall that there is a special combinator for representing the action of creation of a new behaviour

CREATED: behaviour $\rightarrow$ act.

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

functional combinators

for every srt $\in$ SORTS

- $\lambda \Box . \Box$: srt-var $\times$ behaviour $\rightarrow$ funct(srt,behaviour) (λ- abstraction)

The elements of sort funct(srt,behaviour) represent the (partial) functions from elements of sort srt into behaviours; while the elements of sort srt-var represent in some way the "variables of type srt". There is also an operator which embeds these "variables" into the elements of srt

- Srt_Var: srt-var $\rightarrow$ srt

and various combinators for expressing the elements of srt-var.
Notation: for every term of sort srt-var x Srt_Var(x) is simply written x; every string of lower case letters corresponds to a term of sort srt-var.

- **if** $\Box$ **then** $\Box$ **else** $\Box$: bool $\times$ srt $\times$ srt $\rightarrow$ srt (conditional)
- $\Box(\Box)$: funct(srt,behaviour) $\times$ srt $\rightarrow$ behaviour (application)
- **rec** $\Box . \Box$: srt-behaviour-fid $\times$ funct(srt,behaviour) $\rightarrow$ funct(srt,behaviour) (recursive functions constructor)

The elements of sort srt-behaviour-fid represent in some way identifiers of functions of type funct(srt,behaviour); also in this case there is an embedding operation

- Srt_Behaviour_Fid: srt-behaviour-fid $\rightarrow$ funct(srt,behaviour)

and various combinators for expressing the elements of sort srt-behaviour-fid.
Notation: for every term of sort srt-behaviour-fid x Srt_Behaviour_Fid(x) is simply written x; every string of lower case letters represents a term of sort srt-behaviour-fid.
rec fi . funct(fi) represents a function corresponding to a fixpoint of the functional λfi . funct(fi); that fixpoint is defined by the usual rewriting rule **rec** fi . funct(fi) $=$ (λfi . funct(fi))(**rec** fi . funct(fi)).
All these combinators are operators of the specification FUNCT(DATA<srt>,BEHAVIOUR) as introduced in section 1.1. and formally defined in [ARW2]. (If A is a specification and srt a sort of A, then A<srt> indicates that srt is now the main sort of A.)

<u>fixpoint combinators</u>

for every natural number $n \geq 1$

- $\mathbf{fix}_n$: $\text{funct}(\underbrace{\text{prod(behaviour,...,behaviour)}}_{n\ \text{times}}, \underbrace{\text{prod(behaviour,...,behaviour)}}_{n\ \text{times}}) \rightarrow \underbrace{\text{prod(behaviour,...,behaviour)}}_{n\ \text{times}}$

where the elements of sort $\underbrace{\text{prod(behaviour,...,behaviour)}}_{n\ \text{times}}$ are n-tuples of behaviours; moreover on these n-tuples the component selection operations and a constructor operation are defined:

$1 \leq i \leq n$ Sel_i : prod(behaviour,...,behaviour) → behaviour

<□,...,□> behaviour × ... × behaviour →prod(behaviour,...,behaviour).

Considering for simplicity the case n=1, $\mathbf{fix}_1$bhfunct represents a behaviour whose activity is the same activity of bhfunct($\mathbf{fix}_1$bhfunct). These combinators permit to represent behaviours with nonterminating activities and sets of mutually recursive behaviours. For example, $\mathbf{fix}_1\ \lambda\, x.\ a\ \Delta\ x$ represents the behaviour which goes on forever performing the action a. It is important to note that the **fix** combinators are total and that the above operational characterization allows to define completely the behaviours represented by them; moreover they are truly fixpoints, since we have that

$\mathbf{fix}_n$bhfunct = bhfunct($\mathbf{fix}_n$bhfunct). For example $\mathbf{fix}_1\ \lambda\, x\ .\ x$ is defined and represents the behaviour unable to perform any activity, which will be indicated also by **stop**.

<u>nondeterministic choice</u>

for every srt ∈ SORTS

- $\mathbf{choose}_{srt}$ □: funct(srt,behaviour) → behaviour

$\mathbf{choose}_{srt}$ bhfunct represents the behaviours which can nondeterministically behave as specified by bhfunct(t_0) for every term of sort srt t_0.

The importance and relevance of these combinators for representing behaviour subcomponents of concurrent systems should be clear (see, eg [M1, M2]). Following Milner's notations (see, eg [M2]) we would write these combinators as $\underset{t \in SRT}{+}\ bh(t)$, where SRT is a set and bh(t) is a behaviour expression parameterized on t (i.e. an expression of type behaviour with a free variable t of type SRT). Here we are working in a fully algebraic setting, where the elements of SRT are defined by means of an abstract data type with a sort srt and hence $\underset{SRT}{+}$ must be an algebraic operation. The solution we have adopted is to consider a combinator $\mathbf{choose}_{srt}$ applied to functions from elements of sort srt into behaviours; thus the parameterized dependence of bh(t) on t is formally expressed by means of a term of sort funct(srt,behaviour), whose elements correspond to functions from elements of sort srt into behaviours.

Hence $\underset{t \in SRT}{+}\ bh(t)$ will be written $\mathbf{choose}_{srt}\ \lambda t\ .\ bh(t)$.

Notation: $\mathbf{choose}_{srt}\ \lambda\, t\ .\ bh(t)$ is also written **choose** t : srt **in** bh(t) .

Our nondeterministic choice is neither local nor global; it could be local or global depending on the various alternatives:

- if for every term t_0 of sort srt the first-step actions of $bh(t_0)$ correspond to interactions with the other behaviours or the global object, then we have global nondeterminism
  (eg for bh = **choose** n: nat **in** RECnFROMpid Δ **stop**, where we use an infix notation for the receive action REC □FROM □, if the behaviour named pid can send the natural number 1, then bh will choose the alternative REC1 FROMpid Δ **stop**);
- if for every term t_0 of sort srt the first-step actions of $bh(t_0)$ are all internal actions, then we have local nondeterminism (eg if bh = **choose** n: nat **in** TAU Δ SEND(n) Δ **stop**, then bh can choose one of the alternatives independently from the external context).

<u>sequential composition of behaviours</u>

for every srt ∈ SORTS

- $\mathbf{def}_{srt}$ □ **in** □: behaviour × funct(srt,behaviour) → behaviour
- $\mathbf{return}_{srt}$ □: srt → behaviour

The activity of $\mathbf{def}_{srt}$ bh **in** bhfunct consists of the activity of bh until it terminates, followed by the activity of $bhfunct(t_0)$ if bh terminates returning a value t_0 of sort srt and $bhfunct(t_0)$ is defined; $\mathbf{return}_{srt}$ t_0 represents the final state of a behaviour which has terminated its activity returning t_0.

The construct $\mathbf{def}_{srt}$ bh **in** bhfunct is a very general and powerful form of sequential composition because it allows also the preceding behaviour to pass some information to the following one; moreover there is also the possibility of (conditionally) escaping the following behaviour; if bh terminates returning a value t_1 of sort $srt_1 \neq srt$, then the following behaviour represented by bhfunct will not be executed.

In subsection 2.4 we show the use of these combinators for defining a variety of derived combinators.

Notation: $\mathbf{def}_{srt}$ bh **in** λ t . bh'(t) is also written $\mathbf{def}_{srt}$ t = bh **in** bh'(t); $\mathbf{return}_{null}$ Null is also written **skip** (**skip** represents a null behaviour unable to perform any activity).

<u>multilevel structuring combinators</u>

- **i-enclose**: state → behaviour (for enclose concurrent <u>i</u>nductive subcomponent)
- **n-enclose**: sstate → behaviour (for enclose concurrent <u>n</u>oninductive subcomponent)

The elements of sort state represent the states of the system SYST; while the elements of sort sstate represent the states of the concurrent systems SUB-SYST taken as parameter. These two combinators are used for representing multilevel (structured) concurrent systems; the first for the case in which the internal concurrent structure of the behaviours is the same of the whole system (see eg CCS, SCCS), the second when the internal structure is given by means of the parameter SUB-SYST.

The term **i-enclose**(st) represents a behaviour which is internally structured as the concurrent system represented by st and its activity is determinated by the activity of st. Precisely if st can perform some transition labelled by a becoming st', then also **i-enclose**(st) can perform a transition labelled by a becoming **i-enclose**(st') and these are all the transitions of **i-enclose**(st). Thus if the enclosed system st is unable to perform any activity also **i-enclose**(st) is unable to perform any activity; moreover if st represents a correct final state (all the behaviour subcomponents are equal to **skip**), then **i-enclose**(st) = **skip**. Note that this last property allows to compose sequentially concurrent

subcomponents with behaviours. Analogously for **n-enclose** .

If the calculus include the combinator **i-enclose**, since now some transitions (with external flags) of the system may become also transitions, via **i-enclose**, at the behaviour level, then the external flags of SYST must coincide with the behaviour atomic actions (extflag = act); clearly also the external flags of SUB-SYST must coincide with the behaviour atomic actions.

PARALLEL COMBINATOR

The calculus has a combinator which taken some behaviours (a multiset of) and a global object returns a term representing the concurrent system of the class, whose subcomponents are those behaviours and that global object:

- **par**: mset(behaviour) × gobj →state

Notation: a term having form **par**({bh_1,...,bh_n},go) is usually written bh_1|...|bh_n|go to suggest the fact that the various subcomponents are in parallel.

2.3 *Formal definition of a calculus*

First we give the specification of the transition system BH-SYST defining the behaviours and then of the transition system of the whole calculus (SYST).

BEHAVIOURS

Remember that the paramenter SUB-SYST is an algebraic transition system (with transition relation □~~□~~>□: sstate × act × sstate → bool) and that □==□==>□: state × act × state → bool will be the transition relation of SYST (here we consider a calculus including the **i-enclose** combinator, thus act=extflag).

BEHAVIOUR =

enrich $+_{srt \in SORTS}$ FUNCT(DATA(BEHAVIOUR)<srt>, BEHAVIOUR) +

$+_{n \geq 1}$ FUNCT(PROD($\underbrace{\text{BEHAVIOUR,...,BEHAVIOUR}}_{n \text{ times}}$), PROD($\underbrace{\text{BEHAVIOUR,...,BEHAVIOUR}}_{n \text{ times}}$)) +

SUB-SYST + NULL + STATE **by**

sorts behaviour

opns

□Δ□: act × behaviour → behaviour

{ fix_n: funct(prod($\underbrace{\text{behaviour,...,behaviour}}_{n \text{ times}}$),prod($\underbrace{\text{behaviour,...,behaviour}}_{n \text{ times}}$)) → prod($\underbrace{\text{behaviour, ..., behaviour}}_{n \text{ times}}$)) | n≥1 }

{ **choose**$_{srt}$ □: funct(srt,behaviour) → behaviour,

def$_{srt}$ □ **in** □: behaviour × funct(srt,behaviour) → behaviour,

return$_{srt}$ □: srt → behaviour | srt ∈ SORTS }

i-enclose: state → behaviour

n-enclose: sstate → behaviour

seed: → behaviour

axioms "all total"

where

NULL = **sorts** null **opns** Null : $\to$ null **axioms** D(Null)

STATE = **enrich** MSET(BEHAVIOUR) + DATA(BEHAVIOUR) **by**
    sorts state
    opns par: mset(behaviour) $\times$ gobj $\to$ state
    axioms seed|bhms|go = bhms|go.

Note that BEHAVIOUR and STATE are two algebraic specifications defined in a mutually recursive way (see Appendix 1); note also how DATA(X) is recursively instantiated on BEHAVIOUR, so that the behaviours become parts of the data type.

The behaviours will be represented by terms of sort behaviour of the above specification BEHAVIOUR and their combinators will be operations of the same specification. Notation: we recall the abbreviations used: **stop** stands for $\mathbf{fix}_1\ \lambda x\ .\ x$ and **skip** stands for $\mathbf{return}_{null}$Null.

BH-SYST =
  enrich BEHAVIOUR + SYST **by**
    opns $\square \xrightarrow{\square} \square$: behaviour $\times$ act $\times$ behaviour $\to$ bool
      CREATED: behaviour $\to$ act
    axioms
      D(CREATED(bh))
      $a\ \Delta bh \xrightarrow{a} bh$
    { $\mathbf{fix}_n$ bhprodfunct = bhprodfunct($\mathbf{fix}_n$ bhprodfunct) | $n \geq 1$}
    { bhfunct(t) $\xrightarrow{a}$ bh' $\supset$ $\mathbf{choose}_{srt}$ bhfunct $\xrightarrow{a}$ bh'
      $\mathbf{def}_{srt}$ ($\mathbf{return}_{srt}$ t) **in** bhfunct = bhfunct(t)
      bh $\xrightarrow{a}$ bh' $\supset$ $\mathbf{def}_{srt}$ bh **in** bhfunct $\xrightarrow{a}$ bh'
     { $\mathbf{def}_{srt}$ ($\mathbf{return}_{srt1}$ t_1) **in** bhfunct = $\mathbf{return}_{srt1}$ t_1
      Isfree_{srt}(t,bhfunct) = false $\supset$
      $\mathbf{def}_{srt1}$ ($\mathbf{choose}_{srt}$ λt . bh(t)) **in** bhfunct = $\mathbf{choose}_{srt}$ λt .($\mathbf{def}_{srt1}$ bh(t) **in** bhfunct) |srt1 $\in$ SORTS, srt1$\neq$srt}
                                    | srt $\in$ SORTS}
    { **i-enclose**(skip|...|skip|go) = **skip**, **n-enclose**(skip|...|skip|bgo) = **skip** | n $\geq$ 0 }
                    (n times)   (n times)
      st ==a==>st' $\supset$ **i-enclose**(st) $\xrightarrow{a}$ **i-enclose**(st')
      sst ~~a~~>sst' $\supset$ **n-enclose**(sst) $\xrightarrow{a}$ **n-enclose**(sst')
      seed $\xrightarrow{CREATED(bh)}$ bh.

<u>Comments.</u> **seed** is an auxiliary combinator used for allowing dynamic creations of new behaviours.

The functional combinators are defined by the various specifications FUNCT(DATABEHAVIOUR<srt>,BEHAVIOUR).

The axioms of BH-SYST give the operational semantics of the various combinators as it was suggested in subsection 2.2. Isfree_{srt}:srt-var $\times$ funct(srt,behaviour) $\to$ bool is an operation of FUNCT(DATA<srt>,BEHAVIOUR); Isfree_{srt}(x,bhfunct) = true iff the "variable of type srt" x occurs freely in bhfunct. The condition Isfree_{srt}(t,bhfunct) = false in the axiom about **def** and **choose** is not restrictive at all, because on the functions algebraically defined the α-rule holds and there exist infinite

different elements of sort srt-vart. End of comments.

THE CALCULUS

In order to have a full calculus we need to define the synchronization, parallelism and monitoring steps. We recall that this is done by means of a parameterized specification SMoLCS-SYST(PROC-SYST), given as a calculus parameter, where PROC-SYST stands for the algebraic transition system of the component processes. Hence the full calculus will be here the one associated to the algebraic transition system SYST defined as follows:

SYST = SMoLCS-SYST(BH-SYST).

2.4 *Examples*

THE FORMAL DEFINITION OF ADA

Here we show how one of our calculi, denoted by ***AC***, could be used for describing the underlying concurrent model of Ada programs, used in [CRAI-DDC] for giving a formal semantics to Ada.

In this case the parameters are defined as follows:

- The parameter DATA becomes now ADATA(X) = GLOBAL-INF(X) + ACT(X) + LOCAL-INF(X), where GLOBAL-INF(X), ACT(X) (with the operations CREATE, CREATED: behaviour → act) and LOCAL-INF(X) are large and complex specifications representing respectively the global object, the behaviour actions and the data handled locally by behaviours; in this case the sort extflag coincides with act. Recall moreover that X will be instantiated as the specification of behaviours, which corresponds roughly here to Ada tasks.
- In AC the behaviours can interact between them only by reading and updating the global object; contemporaneous behaviour accesses to the global object are allowed if and only if they can also be performed sequentially in some order; moreover there is no form of global control on the behaviour actions. These assumptions are formalized by the following parametric system A-SMoLCS-SYST(PROC-SYST) defined following the SMoLCS three steps schema; where the parameter PROC-SYST will be instantiated with the algebraic transition system giving the operational semantics of behaviours.
- AC is a one-level concurrent system, i.e. there are no behaviours which are in turn concurrent systems themselves; thus in this case we do not need other parameters.

Here we givew the definition of A-SMoLCS-SYST(PROC-SYST).

synchronization

A-SSYST(PROC-SYST)=

enrich PROC-SYST **by**

opns $\square =\square \Rightarrow \square$: state × act × state → bool

axioms Cond(a,go) = true ∧ $bh \xrightarrow{a} bh'$ ⊃ $bh|go \xRightarrow{a} bh'|Transf(a,go)$

$bh \xrightarrow{CREATE(bh_1)} bh'$ ∧ **seed** $\xrightarrow{CREATED(bh_1)} bh_1$ ∧

Cond(CREATE(bh_1),go) = true ⊃

bh| **seed**|go $\xRightarrow{CREATE(bh_1)}$ bh'|bh_1|Transf(CREATE(bh_1),go).

Cond: act × gobj → bool and Transf: act × gobj → gobj are two operations of the specification ACT.

parallelism

A-PSYST(PROC-SYST) =
 enrich A-SSYST(PROC-SYST) **by**
  opns □//□ : act × act → act
  axioms $a_1 // a_2 = a_2 // a_1$ $a_1 //(a_2 // a_3) = (a_1 // a_2) // a_3$
   $<bhms_1|go> ==a_1==> <bhms_1'|go_1> \wedge <bhms_2|go> ==a_2==> <bhms_2'|go_2> \wedge$
   $<bhms_2|go_1> ==a_2==> <bhms_2'|go'> \supset <bhms_1|bhms_2|go> ==a_1//a_2==> <bhms_1'|bhms_2'|go'>$.

The condition part of the above axiom requires that the parallel action labelled by a_2 can be executed after the one labelled by a_1.

monitoring

A-SMoLCS-SYST(PROC-SYST) =
 enrich A-PSYST(PROC-SYST) **by**
  opns □==□==>□ : state × act × state → bool
   Ext: act → act
  axioms st ==a==> st' ⊃ st ==Ext(a)==> st'
   $Ext(a_1 // a_2) = Ext(a_1) // Ext(a_2)$
   Ext-Ax,

where Ext-Ax is a set of axioms defining the operation Ext having form cond ⊃ Ext(a) = a or cond ⊃ Ext(a) = TAU.

The functional combinators of *AC* have been proved very useful in the Ada Formal Definition for expressing, for example, subprograms (Ada procedures and functions), task types and several other kinds of denotations.

Moreover, in order to improve readability, other combinators have been introduced and we show how they can be derived from those of *AC*.

sequential composition without value passing

- □ **;** □: behaviour × behaviour → behaviour
- **nil**: → behaviour

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 the final state of bh_1 is **nil**. Formally

$bh_1 \xrightarrow{a} bh_1' \supset bh_1 ; bh_2 \xrightarrow{a} bh_1' ; bh_2$ **nil ;** bh = bh.

These combinators can be derived in *AC* as follows (we indicate with ≡ equality by definition):

$bh_1 ; bh_2 \equiv$ **def**$_{null}$ $n = bh_1$ **in** bh_2 **nil** ≡ **skip** ≡ **return**$_{null}$ Null

and they have the properties listed above; indeed

$bh_1 \xrightarrow{a} bh_1' \supset bh_1 ; bh_2 \equiv$ **def**$_{null}$ $n = bh_1$ **in** $bh_2 \xrightarrow{a}$ **def**$_{null}$ $n = bh_1'$ **in** $bh_2 \equiv bh_1' ; bh_2$,

nil ; bh ≡ **def**$_{null}$ n = (**return**$_{null}$ Null) **in** bh = (λ n . bh)(Null) = bh.

rec,rec trap,exit

- **trap** □ **in** □: map(label,behaviour) × behaviour → behaviour
- **exit** □: label → behaviour

where label and map(label,behaviour) are sorts of ADATA.

The activity of the behaviour **trap** lmap **in** bh consists of the activity of bh; moreover if bh terminates performing an **exit** to the label l and l belongs to the domain of lmap, then the activity goes on as specified by lmap(l); otherwise the **exit** is propagated to some outer trap construct. These combinators are suggested by VDM combinators introduced for giving the so called direct semantics (see [BJ,AR2]).

Formally i) $l \in dom(lmap) = true \supset$ **trap** lmap **in exit** l = lmap(l)

ii) $l \in dom(lmap) = false \supset$ **trap** lmap **in exit** l = **exit**(l)

iii) bh $\xrightarrow{a}$ bh' $\supset$ **trap** lmap **in** bh $\xrightarrow{a}$ **trap** lmap **in** bh' .

These combinators can be derived in *AC* as follows:

trap lmap **in** bh $\equiv$ $\mathbf{def}_{label}$ l = bh **in** (**if** $l \in$ dom(lmap) **then** lmap(l) **else** $\mathbf{return}_{label}$ l)

exit l $\equiv$ $\mathbf{return}_{label}$ l.

- **rec trap** □ **in** □: map(label,behaviour) × behaviour → behaviour

rec trap is similar to the **trap**, except that axiom i) is replaced by

i') $l \in dom(lmap) = true \supset$ **rec trap** lmap **in exit** l = **rec trap** lmap **in** lmap(l).

It can be derived in *AC* as follows

rec trap $[l_1 \to bh_1, ..., l_n \to bh_n]$ **in** bh $\equiv$ **trap** $[l_1 \to bh_1', ..., l_n \to bh_n']$ **in** bh

where for every $1 \le i \le n$ $bh_i' = Sel_i(bh')$ and

bh' = $\mathbf{fix}_n\, \lambda x.$ <**trap** $[l_1 \to Sel_1(x),...,l_n \to Sel_n(x)]$ **in** bh_1,

. . .

trap $[l_1 \to Sel_1(x),...,l_n \to Sel_n(x)]$ **in** bh_n>.

It is easy to see that the derived combinators have the properties i), ii), iii) and i'), ii), iii) respectively.

SEQUENTIAL CONSTRUCTS

Here we show how it is possible to enrich our calculi with the usual sequential constructs, deriving them by the calculi combinators.

We assume that each process has a local store whose states are represented by elements of sort store = map(loc,value); these processes will be represented by elements of sort proc = funct(store,behaviour) and a system whose subcomponents are the processes $proc_1,...,proc_n$ will be represented by $proc_1$(Empty_Map)|...|$proc_n$(Empty_Map)|go (Empty_Map represents the initial empty state of the local store).

The derived combinators are:

- □:= □: loc × expression →proc

(l := exp $\equiv$ λ st. TAU Δ $\mathbf{return}_{store}$ st[Eval(exp,st)/l]),

where Eval: expression × store → value is an operation of DATA.

- If □ Then □ Else □: expression × proc × proc →proc

(If exp Then pr_1 Else pr_2 $\equiv$ λ st. TAU Δ **if** Eval(exp,st) **then** pr_1(st) **else** pr_2(st)).

Note that for simplicity we consider expressions without side effects.

- □; □: proc × proc →proc (pr_1; pr_2 $\equiv$ λ st. $\mathbf{def}_{store}$ pr_1(st) **in** pr_2).
- While □ > 0 Do □: expression × proc →proc

(While exp Do pr $\equiv$ **rec** wh . λ st. TAU Δ **if** Eval(exp,st) > 0 **then** $\mathbf{def}_{store}$ pr(st) **in** wh

else $\mathbf{return}_{store}$ st).

- □ : act →proc (a $\equiv$ λ st. a Δ $\mathbf{return}_{store}$ st).
- $Choose_{srt}$ □: funct(srt,proc) →proc ($Choose_{srt}$ λ v. pr(v) $\equiv$ λ st. $\mathbf{choose}_{srt}$ λ v. (pr(v))(st)).

3 PROPERTIES OF COMBINATORS

Here we study the properties of the combinators introduced in the preceding section. Some of these properties are just equalities provable from the given specification; for other deeper properties we have to consider equivalences with respect to some observations. The most basic form of observation consists in observing the actions of a behaviour, which leads to the well known notion of strong (bisimulation) equivalence of Milner and Park. For our calculi we need to generalize that notion, since our flags may include behaviours as subterms; moreover we would like to equate functional terms by extensionality. Since at the present stage of our investigation the theory related to such generalization looks a bit complicate we defer the presentation of the full theory to a more technical paper; hence we prefer to present the properties of combinators for subcalculi, in which the flags cannot have behaviours as subterms. But the properties we show do hold in the general unrestricted case and hence they give a rather good understanding of the properties of the calculi.

3.1 *Strong equivalence properties of behaviour combinators*

Let BH-SYST indicate the transition system (with transition relation
$\square \xrightarrow{\square} \square$:behaviour × act × behaviour →bool) defining the behaviours of one of our calculi.
In the following we consider only calculi of behaviours in which the flags do not have behaviour subterms, formally calculi such that the parameter DATA (see section...) is not a specification parameterized on behaviours, and such that BH-SYST is image finite. (A transition system with transition relation $\longrightarrow$ is said <u>image finite</u> iff for all states s and flags f the set $\{s' \mid s \xrightarrow{f} s'\}$ is finite.) By <u>strong equivalence</u> we mean strong bisimulation equivalence, as introduced in [M2].
From the beginning we have to face an interesting problem: in our calculi of behaviours $\mathbf{return}_{srt}$ v and **stop** are two normal states, i.e. behaviours without action capabilities, and hence they would be equated in the strong equivalence associated to the behaviour transition system. But inserted in the context
$\mathbf{def}_{srt}$ [x] **in** bhfunct they wuold produce two behaviours which are not strongly equivalent and hence the strong equivalence wuold not be a congruence. Since clearly we are interested in a strong equivalence which is also a congruence, we simply distinguish the two behaviours by considering the strong equivalence associated to a modified behaviour transition system, obtained by BH-SYST by adding a set of (dummy) transitions defined by $\mathbf{return}_{srt}\, t \xrightarrow{RETURN_{srt}(t)} \mathbf{stop}$.
Thus we indicate by ~ the strong equivalence w.r.t. the new transition system of behaviours obtained by adding the above transitions.
We give only some hints to the proofs, that will appear in a full version elsewhere.
We can now give a basic result for behaviours without **i-enclose** and **n-enclose** combinators; in the next subsection we will extend it to the general case.
As Milner in [M2] extends ~ from agents to expressions, we extend ~ from behaviours to functions returning behaviours; given two terms f_1 and f_2 of sort funct(srt,behaviour)

$f_1 \sim f_2$ iff for all terms t_1, t_2 of sort srt $t_1 = t_2$ implies $f_1(t_1) \sim f_2(t_2)$.

Theorem 3. ~ is a congruence on behaviours (without the **i-enclose** and **n-enclose** combinators).

Proof. - $bh_1' \sim bh_2'$ and $a_1 \equiv a_2$ implies $a_1 \Delta bh_1' \sim a_2 \Delta bh_2'$. Obvious.

- For all srt ∈ SORTS, $bhfunct_1 \sim bhfunct_2$ implies **choose**$_{srt}$ $bhfunct_1$ ~ **choose**$_{srt}$ $bhfunct_2$. Obvious.

- $bhfunct_1 \sim bhfunct_2$ implies **fix**$_1$ $bhfunct_1$ ~ **fix**$_1$ $bhfunct_2$. Analogously to the proof of Proposition 4.6 of [M2].

- For all srt ∈ SORTS, $bh_1' \sim bh_2'$ and $bhfunct_1 \sim bhfunct_2$ implies

 def$_{srt}$ bh_1' **in** $bhfunct_1$ ~ **def**$_{srt}$ bh_2' **in** $bhfunct_2$.

We show that

R = {< **def**$_{srt}$ bh' **in** bhfunct', **def**$_{srt}$ bh'' **in** bhfunct''> | bh' ~ bh'' and bhfunct' ~ bhfunct''} ∪ Id,

where Id indicates the identity relation, is a bisimulation up to ~ (i.e. $\sim R \sim$ is a bisimulation).

If R is a bisimulation up to ~, then $R \subset \sim R \sim \subset \sim$.

Let $<bh_1,bh_2>$ be an element of R, we prove, by cases, that

if $bh_1 \xrightarrow{a} bh_1'$, then $bh_2 \xrightarrow{a} bh_2'$ and $bh_1' \, R \sim bh_2'$.

° $bh' \xrightarrow{a} bh_1''$, bh_1' = **def**$_{srt}$ bh_1'' **in** bhfunct' and a ≠ $RETURN_{srt}(..)$.

By the hypothesis $bh'' \xrightarrow{a} bh_2''$ and $bh_1'' \sim bh_2''$, thus $bh_2 \xrightarrow{a} bh_2'$,

bh_2'= **def**$_{srt}$ bh_2'' **in** bhfunct'' and $bh_1' \, R \sim bh_2'$.

° bh' = **return**$_{srt}$ t_1 and $bhfunct'(t_1) \xrightarrow{a} bh_1'$.

By the hypothesis bh'' = **return**$_{srt}$ t_2 and $t_1 \equiv t_2$; $bhfunct'(t_1) \sim bhfunct''(t_2)$ implies $bhfunct''(t_2) \xrightarrow{a} bh_2'$ and $bh_1' \sim bh_2'$; thus $bh_2 \xrightarrow{a} bh_2'$ and $bh_1' \, R \sim bh_2'$.

° bh' = **return**$_{srt1}t_1$ with srt1≠srt. Thus $bh_1 \equiv$ **return**$_{srt1}t_1 \xrightarrow{RETURN_{srt1}(t_1)}$ **stop**.

By hypothesis bh'' = **return**$_{srt1}t_2$ with $t_1 \equiv t_2$, thus $bh_2 \equiv$ **return**$_{srt1}t_2 \xrightarrow{RETURN_{srt1}(t_2)}$ **stop**. □

Proposition 4. (**def/return** properties). For every srt ∈ SORTS

1) **def**$_{srt}$ (**return**$_{srt}$ t) **in** bhfunct ~ bhfunct(t).

2) For every srt1 ∈ SORTS such that srt1 ≠ srt **def**$_{srt}$ (**return**$_{srt1}$ t_1) **in** bhfunct ~ **return**$_{srt1}$ t_1.

3) For every srt1 ∈ SORTS

 $Isfree(t_1,bhfunct) \equiv false \supset$

 def$_{srt}$ (**choose** t_1: srt1 **in** $bh(t_1)$) **in** bhfunct ~ **choose** t_1: srt1 **in** (**def**$_{srt}$ $bh(t_1)$ **in** bhfunct).

4) **def**$_{srt}$ (a Δ bh) **in** bhfunct ~ a Δ (**def**$_{srt}$ bh **in** bhfunct) .

Proof. 1), 2) and 3) Obvious, because ≡ implies ~. 4) obvious. □

Proposition 5. (choose properties)

1) For every n,m≥1, $srt_1,...,srt_n,srt_1',...,srt_m'$∈ SORTS

[for all terms $t_1,...,t_n$ of sort $srt_1,...,srt_n$ respectively

there exist $t_1',...,t_m'$ terms of sort $srt_1',...,srt_m'$ respectively such that $bh_1(t_1,...,t_n) \sim bh_2(t_1',...,t_m')$]

and

[for all terms $t_1',...,t_m'$ of sort $srt_1',...,srt_m'$ respectively

there exist $t_1,...,t_n$ terms of sort $srt_1,...,srt_n$ respectively such that $bh_2(t_1',...,t_m') \sim bh_1(t_1,...,t_n)$]

implies

choose t_1: srt_1 **in** ... **choose** t_n: srt_n **in** $bh_1(t_1,...,t_n)$ ~
choose t_1': srt_1' **in** ... **choose** t_m': srt_m' **in** $bh_2(t_1',...,t_m')$.

2)(Idempotence) For every srt ∈ SORTS

[for all terms t of sort srt bhfunct(t) ~ bh] implies $\mathbf{choose}_{srt}$ bhfunct ~ bh.

Proof. Obvious from the definition of ~ . □

For simplicity we consider only the combinator $\mathbf{fix}_1$, i.e. the unary fixpoint combinator.
For expressing the properties of the $\mathbf{fix}_1$ combinator we need the following definitions and lemmas.
der bh indicates the derivation tree of bh, i.e. the labelled tree associated to bh in the transition system; given a derivation tree tr, $|tr|_n$ indicates the truncation of tr at depth n. Given a term f of sort funct(srt_1,srt_2), $f^n(t)$ indicates f applied n times to t.
Similarly as in [M1] for CCS, we define that a variable x of type behaviour is guarded in bh (i.e. x is preceded in bh by a Δ..., for some action a); we omit the trivial definition by induction on the structure of behaviours.

Lemma 1. (basic **fix** lemma) If the variable x is guarded in bh(x), then given A = $\mathbf{fix}_1$ λ x . bh(x) and for m ≥ 1 A_m= (λ x . bh(x))^m(**stop**) , we have that

for all n ≥ 1 $|der\ A|_n = |der\ A_n|_n = |der\ A_{n+q}|_n$ (for all q ≥ 1).

Proof. By arithmetic induction on n. □

Lemma 2. (**fix**-context lemma) For all terms bh(y) of sort behaviour with a hole of sort behaviour, if bhfunct = λ x . $bh_1(x)$ with x guarded in $bh_1(x)$, A = bh($\mathbf{fix}_1$ bhfunct) and for all p ≥ 1 A_p = bh(bhfunct^p(**stop**)), then we have that for all n ≥ 1 $|der\ A|_n = |der\ A_n|_n$.

Proof. By structural induction on bh(y). □

Proposition 6. ($\mathbf{fix}_1$ properties)

1) For all terms $bh_1(y)$, $bh_2(y)$ of sort behaviour with a hole of sort behaviour,
for all bhfunct = λx. bh(x) with x guarded in bh(x)
[for all n ≥ 1 $|der\ (bh_1(bhfunct^n(\mathbf{stop})))|_n = |der\ (bh_2(bhfunct^n(\mathbf{stop})))|_n$] implies
bh_1($\mathbf{fix}_1$ bhfunct) ~ bh_2($\mathbf{fix}_1$ bhfunct) .

2) If x is guarded in bh(x), then
Isfree(x,bhfunct) = false implies
$\mathbf{def}_{srt}$ ($\mathbf{fix}_1$ λx . bh(x)) **in** bhfunct ~ $\mathbf{fix}_1$ λx . ($\mathbf{def}_{srt}$ bh(x) **in** bhfunct).

Proof. Obvious, by Lemma 2. □

3.2 *Strong equivalence properties of parallel combinator*

Here we use **~** (~ bold) to indicate the strong extensional equivalence of the algebraic transition system SYST (with transition relation □==□==>□:state × extflag × state →bool) defining one of our calculi

and P-SYST and S-SYST indicate respectively the systems defined by the parallel and synchronous steps (recall that SYST has been defined following the three steps SMoLCS methodology).

Also in this case we need to distinguish the normal states of SYST and as in the previous section to do this we add some transitions to SYST; precisely

skip|...|**skip**|go $\xrightarrow{\text{CORRECT}}$ **stop**|go;

which allow to distinguish the correct terminal states from the incorrect ones..

Proposition 7. (| properties)

1) For every $bh^1{}_1|...|\ bh^1{}_n|go,\ bh^2{}_1|...|\ bh^2{}_n|go \in W_{Sig(SYST)}|_{state}$ such that $bh^1{}_1 \sim bh^2{}_1, ..., bh^1{}_n \sim bh^2{}_n$ we have that $bh^1{}_1|...|\ bh^1{}_n|go \sim bh^2{}_1|...|\ bh^2{}_n|go$;

2) for every $bhms|go \in W_{Sig(SYST)}|_{state}$ **seed**|bhms|go ~ bhms|go ;

3) for every $bhms|go \in W_{Sig(SYST)}|_{state}$, for every $bh \in W_{Sig(SYST)}|_{behaviour}$ such that bh ~ **skip**

bh|bhms|go ~ bhms|go.

Proof. 1) By Lemma 3. 2) and 3) Obvious. □

Lemma 3.(Monitoring step)

For every $bh^1{}_1|...|\ bh^1{}_n|go,\ bh^2{}_1|...|\ bh^2{}_n|go \in W_{Sig(SYST)}|_{state}$,

for every $extf \in W_{Sig(SYST)}|_{extflag}$ such that $bh^1{}_1 \sim bh^2{}_1, ..., bh^1{}_n \sim bh^2{}_n$ we have that

i) for every $bh^1{}_1'|...|\ bh^1{}_n'|go' \in W_{Sig(SYST)}|_{state}$

SYST |– $bh^1{}_1\ |...|\ bh^1{}_n|go$ ==extf==> $bh^1{}_1'|...|\ bh^1{}_n'|go'$ implies

there exist $bh^2{}_1'|...|\ bh^2{}_n'|go' \in W_{Sig(SYST)}|_{state}$ such that

SYST |– $bh^2{}_1|...|\ bh^2{}_n|go$ ==extf==> $bh^2{}_1'|...|\ bh^2{}_n'|go'$ and $bh^1{}_1' \sim bh^2{}_1', ..., bh^1{}_n' \sim bh^2{}_n'$;

ii) converse of i).

Proof. By Lemma 4, recalling that in a SMoLCS system the monitoring decision depends only on the possible actions of the behaviours and not on their states. □

Lemma 4. (Parallelism step)

For every $bh^1{}_1|...|\ bh^1{}_n|go,\ bh^2{}_1|...|\ bh^2{}_n|go \in W_{Sig(SYST)}|_{state}$,

for every $a \in W_{Sig(SYST)}|_{act}$ such that $bh^1{}_1 \sim bh^2{}_1, ..., bh^1{}_n \sim bh^2{}_n$ we have that

i) for every $bh^1{}_1'|...|\ bh^1{}_n'|go' \in W_{Sig(SYST)}|_{state}$

P-SYST |– $bh^1{}_1\ |...|\ bh^1{}_n|go$ ==a==> $bh^1{}_1'|...|\ bh^1{}_n'|go'$ implies

there exist $bh^2{}_1'|...|\ bh^2{}_n'|go' \in W_{Sig(SYST)}|_{state}$ such that

P-SYST |– $bh^2{}_1|...|\ bh^2{}_n|go$ ==a==> $bh^2{}_1'|...|\ bh^2{}_n'|go'$ and

$bh^1{}_1' \sim bh^2{}_1', ..., bh^1{}_n' \sim bh^2{}_n'$;

ii) converse of i).

Proof. By Lemma 5. □

Lemma 5. Under the same hypotheses of Lemma 3, we have that

i) for every $bh^1{}_1'|...|\ bh^1{}_n'|go' \in W_{Sig(SYST)}|_{state}$

S-SYST $\vdash bh^1{}_1 | \ldots | bh^1{}_n | go \overset{a}{\Longrightarrow} bh^1{}_1' | \ldots | bh^1{}_n' | go'$ implies
there exist $bh^2{}_1' | \ldots | bh^2{}_n' | go' \in W_{Sig(SYST)}|_{state}$ such that
S-SYST $\vdash bh^2{}_1 | \ldots | bh^2{}_n | go \overset{a}{\Longrightarrow} bh^2{}_1' | \ldots | bh^2{}_n' | go'$ and
$bh^1{}_1' \sim bh^2{}_1', \ldots, bh^1{}_n' \sim bh^2{}_n'$;

ii) converse of i).

Proof. By cases on the form of a. □

Proposition 8. (**i-enclose, n-enclose** properties)

1) For every $n \geq 1$ **i-enclose**($\underbrace{\textbf{skip}|\ldots|\textbf{skip}}_{n \text{ times}}|go$) ~**skip** **n-enclose**($\underbrace{\textbf{skip}|\ldots|\textbf{skip}}_{n \text{ times}}|bgo$) ~ **skip**.

2) For srt ∈ SORTS for every $n \geq 0$

$\bigwedge_{1 \leq i \leq n} \text{Isfree}(t, bh_i) = \text{false} \wedge \text{Isfree}(t, go) = \text{false} \supset$

i-enclose((**choose** t: srt **in** bh) $|bh_1|\ldots|bh_n|go$) ~ **choose** t: srt **in**(**i-enclose**(bh $|bh_1|\ldots|bh_n|go$))
n-enclose((**choose** t: srt **in** bh) $|bh_1|\ldots|bh_n|bgo$) ~ **choose** t: srt **in**(**n-enclose**(bh $|bh_1|\ldots|bh_n|bgo$))

3) st ~ st' ⊃ **i-enclose**(st) ~ **i-enclose**(st'), sst ≃ sst' ⊃ **n-enclose**(sst) ~ **n-enclose**(sst')
where ≃ indicates the strong equivalence on the transition system SUB-SYST (parameter of the calculus) representing the noninductive concurrent components.

Proof. Obvious. □

Now we can extend Theorem 3 to all behaviours.

Theorem 3.BIS. ~ is a congruence on behaviours.

Proof. By Theorem 3, Proposition 7 and Proposition 8. □

Conclusion

We have presented a proposal for a family of calculi, which are a partial instantiation of the SMoLCS parameterized schema. The novelties of these calculi lie in their high level of parameterization, in the possibility of defining functional modules and of considering processes just as data types. In this sense we personally see our calculi as a development for high-level specifications of the work started with CCS and SCCS, which we consider basic calculi, much as lambda-calculi are w.r.t. higher level languages.
We are well aware that our presentation here is far from being satisfactory in many respects. We plan to come out with a more explanatory paper with full proofs. We are currently pursuing two directions of interesting research: first we are looking at a nice proof techniques for a generalization of strong equivalence handling labels with behaviours as subterms and including extensionality; second, we have already explored in part the possibility of calculi where the behaviour labels include, so to speak, the code for the interactions at synchronization, parallelism and monitoring level, while still keeping the full expressive power of SMoLCS specifications; but it is not clear whether this calculus can be elegant and simple enough to be really useful.
Finally we want to emphasize that the family of calculi *AC* we have used in the Ada Formal Definition

project can be seen as an upgrading of the VDM metalanguage Meta IV to handle concurrency and abstract data types.

Appendix 1: *Recursive specifications*

Let SPEC = $\bigcup_{\Sigma \in SIG} \{\Sigma, Ax) \mid Ax$ is a set of positive conditional axioms on $\Sigma\}$, where SIG is the set of all (classes of isomorphic) signatures. A recursive definition of a specification has form

(*) ID = S(ID),

where S is a function from SPEC into SPEC.

After having defined an ordering « on SPEC we can see (*) as defining ID as the least fixpoint of S (if there exists).

Given $S_1=(\Sigma_1,Ax_1)$ and $S_2=(\Sigma_2,Ax_2)$, $S_1 \ll S_2$ iff Σ_1 is a subsignature of Σ_2 and $Ax_1 \subseteq Ax_2$; given $\Sigma_1 = (Sorts_1,Opns_1)$ and $\Sigma_2 = (Sorts_2,Opns_2)$, Σ_1 is a subsignature of Σ_2 iff $Sorts_1 \subseteq Sorts_2$ and $Opns_1 \subseteq Opns_2$.

If S is continuous, then (*) defines the specification ID in the following way: $ID = \text{l.u.b.}_{n \geq 0}\ S^n(ID_\perp)$,

where $ID_\perp$ is equal to the specification with only one sort named id and neither operations nor axioms.

Whenever S is given by composing constant specifications, the parametric specifications, PROD, MAP, MSET and the "+" and "**enrich** ... **by**..." operators, then it is continuous.

Clearly also sets of mutually recursive specifications can be defined in the same way.

Whenever the l.u.b.of the chain $\{S^n(ID_\perp)\}_{n\geq 0}$ is obtained as the k-th step for some k, then the specification can be given in a non recursive way.

Here as an example we report a nonrecursive definition of the specification PROC, defined recursively in subsection 1.1.

PROC = **enrich** NAT+ LOC + BUFID + CHID **by**

sorts	value, proc, instr	
opns	<□, □>: instr × lmem	→ proc
	Emptymap:	→ lmem
	□[□/□]: lmem × loc × value	→ lmem
	□(□): lmem × loc	→ value
	Nil:	→ proc
	Pval: proc	→ value
	Nval: nat	→ value
	{Op: value × ... × value → value | Op: nat × ... × nat → nat ∈ Sig(NAT) }	
	Writebuf, Readbuf: loc × bufid	→ instr
	Send, Rec: loc × chid	→ instr
	Skip:	→ instr
	Start: proc	→ instr
	□ ; □, □ + □: instr × instr	→ instr
	While □ ≠ 0 Do □: loc × instr	→ instr
	Seq-Instr_1: ...	→ instr
	...	
	Seq-Instr_n: ...	→ instr

axioms "all total"

$\{Op(Nval(n_1),...,Nval(n_k)) = Nval(Op(n_1,...,n_k)) \mid Op: nat \times ... \times nat \to nat \in Sig(NAT)\}$

$Eq(l_1,l_2) = true \supset (lm[v/l_1])(l_2) = v$

$Eq(l_1,l_2) = false \supset (lm[v/l_1])(l_2) = lm(l_2)$

$i_1 ; (i_2 ; i_3) = (i_1 ; i_2) ; i_3$ $\quad$ $Skip ; i = i$

$i_1 + i_2 = i_2 + i_1$ $\quad$ $i_1 + (i_2 + i_3) = (i_1 + i_2) + i_3$.

Acknowledgements. We wish to acknowledge the invaluable cooperation of Martin Wirsing in building the foundations of the SMoLCS approach. Moreover we wish to thank all our friends of the Genoa-CRAI group (Alessandro Giovini, Franco Mazzanti, Elena Zucca), who have used, tested and improved our calculi in the Ada FD project. Many thanks also to Ombretta Arvigo for her patient Mac-typing and more generally for her cooperation at any time.

REFERENCES

(LNCS stands for Lecture Notes in Computer Science, Springer Verlag).

[AGMRZ] E.Astesiano, A.Giovini, F.Mazzanti, G.Reggio, E.Zucca, *The Ada challenge for new formal semantic techniques*, in Proc. of the 1986 Ada International Conference, Edinburgh, Cambridge University Press, UK, 1986.

[AMRW] E.Astesiano, G.F.Mascari, G.Reggio, M.Wirsing, *On the parameterized algebraic specification of concurrent systems*, Proc. CAAP '85 - TAPSOFT Conference, LNCS n. 185, 1985.

[AMRZ] E.Astesiano, F.Mazzanti, G.Reggio, E.Zucca, *Applying the SMoLCS specification methodology to the CNET architecture*, CNET - Distribute Systems on Local Network, vol 2, pp. 255-267, ETS Pisa,1985.

[AMRZ1] E.Astesiano, F.Mazzanti, G.Reggio, E.Zucca, *Formal specification of a concurrent architecture in a real project*, Proc. of ACM-ICS'85, North Holland, 1985.

[AR1] E.Astesiano, G.Reggio, *A syntax-directed approach to the semantics of concurrent languages*, in Proc. 10th IFIP World Congress (H.J. Kugler ed.), North Holland,p. 571-576, 1986.

[AR2] E.Astesiano,G. Reggio, *Comparing direct and continuation styles for concurrent languages*, to appear in Proc. STACS 87', LNCS, 1987.

[AR3] E.Astesiano, G.Reggio, *The SMoLCS approach to the formal semantics of programming languages* - A tutorial introduction - to appear in Proc. of CRAI Spring International Conference: Innovative software factories and Ada, 1986.

[ARW1] E.Astesiano, G.Reggio, M.Wirsing, *Relational specifications and observational semantics*, in Proc. of MFCS'86, LNCS n. 233, 1986.

[ARW2] E.Astesiano, G.Reggio, M.Wirsing, *On the algebraic specification of function spaces*, in preparation.

[ARW3] E.Astesiano, G.Reggio, M.Wirsing, *A modular parameterized algebraic approach to the specification of concurrent systems*, in preparation.

[BJ] D.Bjørner, C.B.Jones, *The Vienna development method: The Meta-Language*, LNCS n. 61, 1978.

[BW1] M.Broy, M.Wirsing, *On the algebraic specification of finitary infinite communicating sequential processes,* in Proc. IFIP TC2 Working Conference on "Formal Description of Programming Concepts II", (D. Bjørner ed.), North Holland, 1983.

[BW2] M.Broy, M.Wirsing, *Partial abstract types,* Acta Informatica 18, 1982.

[BW3] M.Broy, M.Wirsing, *Algebraic definition of a functional programming language and its semantic models*, R.A.I.R.O. vol. 17,1983.

[CRAI-DDC] E.Astesiano, C.Bendix Nielsen, N.Botta, A.Fantechi, A.Giovini, P.Inverardi, E. Karlsen, F.Mazzanti, J. Storbank Pedersen, G.Reggio, E.Zucca, Deliverable 7 of the CEC MAP project: The Draft Formal Definition of ANSI/MIL-STD 1815 Ada, 1986.

[H] H.Hussmann, *Rapid prototyping for Algebraic Specifications* RAP *system user's manual,* MIP 8502, Universitat Passau, 1985.

[M1] R.Milner, *A calculus of communicating systems,* LNCS n. 92, 1980.

[M2] R.Milner, *Calculi for synchrony and asynchrony,* TCS 25, 267-310, 1983.

[Mo] F.Morando, *An interpreter for concurrent systems SMoLCS specifications,* Thesis (in italian) University of Genova, Italy, 1986.

[P] G.Plotkin, *A structural approach to operational semantics,* Lecture notes, Aarhus University, 1981.

[SW] D.T.Sannella, M.Wirsing, *A kernel language for algebraic specifications and implementation,* in Proc. Int. Conf. on Foundations of Computation Theory, Borgholm, Sweden, LNCS n.158, 1983.

[W] M.Wirsing, *Structured algebraic specifications: a kernel language,* TCS Vol.42 n. 2, 1986.