A Metalanguage for the Formal Requirement Specification of Reactive Systems *

Egidio Astesiano – Gianna Reggio

Università di Genova – Dipartimento di Informatica e Scienze dell'Informazione
Viale Benedetto XV,3 16132, Genova, Italy
astes, reggio cisi.unige.it

1 Introduction

Various formalisms have been proposed for the specification of software / hardware systems characterized by the possibility of performing some dynamic activities interacting with an external world, called reactive systems; however in the literature sometimes also terms as: concurrent, parallel, distributed, ... systems have been used for pointing out to some particualr features of the systems; here we simply use the term *reactive systems*. Some of these formalisms deal with properties at what we may call design level, when already architectural decisions have been taken and a specification determines essentially one structure, though still at a rather abstract level (we say abstract specifications). Among these, some, like CCS, CSP and variations, are more languages for describing elegant models than specification formalisms; others are instead suitable for expressing, with more or less generality, abstract properties about the static data, like PSF [12], LOTOS [10] and algebraic Petri nets [20] or about both the static data and the concurrent architecture, like SMoLCS [1, 2, 5]. The interested reader may wish to look at [3] for a survey dealing mainly with abstract specifications at the design level.

At the requirement level, the proposed formalisms are dealing with the abstract dynamic properties, i.e. those related to the possible events in a system life, usually classified in safety and liveness properties. There is a literature on formalisms based on temporal, deontic, event logic and others (see e.g. [15, 9, 17], also for references). Now the experience shows that also the structural properties of a system (including the static data) and their relationships with dynamic features of a system are fundamental also at the requirement level; however the specification mechanism should be able to avoid overspecification, not confusing requirements and design.

We intend in this paper to propose an approach, supported by a metalanguage (schema), for dealing, at the requirement level, with both static and dynamic properties.

The approach is based on a specification formalism which, according to the institution paradigm (see [6]), consists in models (semantic structures), sentences or formulae (syntax) and validity (semantics of sentences). A specification is a set of formulae determining a class of models, all those satisfying the formulae. The new and

* This work has been supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of C.N.R. (Italy), Esprit-BRA W.G. COMPASS n. 6112 and by a grant ENEL-SPA/CRA (Milano Italy).

central idea of this paper is the proposed models, that we call entity algebras. Those models can support statements about the structure of reactive systems, dealing with the subcomponents of a system, without referring to detailed structuring combinators, which are essential at the design level, but here would spoil the generality of requirements.

In the first section we give a rather informal presentation of entity algebras, illustrated by a small example. In the second we present the syntax of a specification language and in the third section the notion of validity of a formula and the semantic of a specification; some examples concerning also the application to an industrial test case are reported in the fourth section. The used algebraic notions and notations are reported in Appendix A.

Two important comments are in order. First our metalanguage is rather schematic, in the sense that we can choose various formalisms for expressing the dynamic properties. Here, we give just a set of combinators, taken from the branching time logic, sufficient for expressing some common interesting requirements on reactive systems; for other choices, see e.g. [8] which uses a richer choice of branching time combinators and [17], which presents an event logic, where the properties on the activity of the elements are expressed in terms of relationships among occurrences of non-instantaneosus events. In general, depending on the specific application field (e.g., industrial plants handled by automatisms, software / hardware architectures and so on) one can choose a minimal set of combinators getting a formalism powerful enough but rather simple. The important point is that now the dynamic formulae are "anchored" to a term representing a dynamic element and so we have a formalism where it is possible to express requirements involving properties on the activity either of different systems or of a system and some of its components; for example we can express properties of the form "a new component which can eventually reach a certain situation cannot be added to a system satisfying some condition".

Second, though we do not insist on the more formal aspects, it can be shown formally (not a trivial task) that indeed we get an institution; the interested reader may look at [16, 4, 7] for a more formal presentation of entity algebras and specifications.

2 The Models

2.1 Entity Algebras

Here we introduce our models for reactive systems called *entity algebras* (*entity* is the word that we use for possibly structured dynamic elements).

The models should allow to represent the dynamic activity of the entities. We adopt the well-known and accepted technique which consists in viewing entities as labelled transition trees defined by a labelled transition system (see [13, 14]).

A *labelled transition system* (shortly *lts*) is a triple $(STATE, LAB, \rightarrow)$, where $STATE$ and LAB are sets, whose elements represent respectively the *states* and the *labels* of the system and $\rightarrow \subseteq STATE \times LAB \times STATE$ is the *transition relation*; $(s, l, s') \in \rightarrow$ is said a *transition* and is usually denoted by $s \xrightarrow{l} s'$.

An entity E can be represented by an lts LTS and an initial state $s_0 \in STATE$; then the states of LTS reachable by a sequence of transitions from s_0 represent the intermediate (interesting) situations of the activity of E and the transition relation of LTS the possibilities of E of passing from one situation to another. Note that here a transition $s \xrightarrow{l} s'$ has the following meaning: E in the state (situation) s has the *capability* of passing into the state (situation) s' by performing a transition whose interaction with the external (to E) world is represented by the label l; thus l contains both information on the conditions on the external world for the capability to become effective, and on the transformation of the external world induced by the execution of the transition.

The models should allow to represent the structure of the entities. Usually most of the reactive systems of interest are structured, i.e. are systems where dynamic subcomponents interact among them for determining the activity of the whole system (in general the dynamic subcomponents may be in turn structured systems). Think for example of the Ada programs (whose subcomponents are the tasks, while tasks have no subcomponents, they are simple) or a net of workstations on which UNIX is running (the subcomponents of the nets are the workstations, while the subcomponents of the workstations are the UNIX processes, which may be simple or have as subcomponents other processes). So in general in a system there are different "sorts" (types) of entities, all of them modelled by lts's. In the Ada case there are entities of sort "program" and "task", while in the workstation case of sort "net", "workstation" and "process". Thus the first component of our model is

$$\{LTS_{es}\}_{es \in ES},$$

where ES is the set of the various sorts of entities and for each $es \in ES$

$$LTS_{es} = (STATE_{es}, LAB_{es}, \rightarrow_{es}) \text{ is an lts.}$$

Then the models should allow to know which are the subcomponents of an entity (possibly none) with their sorts and how they are organized to get the whole entity; to this end we have *entity composers*, i.e. partial functions taking as arguments entities, also of different sorts, and returning an entity of a certain sort; precisely an entity composer of arity "$es_1 \times \ldots \times es_n \rightarrow es$", for $n \geq 0$ is:

- when $n > 0$, a partial function

$$Ec\colon STATE_{es_1} \times \ldots \times STATE_{es_n} \rightarrow STATE_{es};$$

- when $n = 0$, a constant

$$Ec \in STATE_{es}.$$

If $e = Ec(e_1, \ldots, e_n)$, then e_1, …, e_n are the *(immediate) subcomponents* of e.

By giving a set of entity composers, we get the complete views of the structure of the entities.

For example, assume $e = Ec(e_1, \ldots, e_n)$, $e_1 = Ec'(e'_1, \ldots, e'_k)$ and that e'_1, …, e'_k, e_2, …,e_n have no subcomponents, i.e. for $i = 1$, …, k $e'_i = Ec'_i$ and for $j = 2$,

$\ldots, n$ $e_j = Ec_j$ with $Ec'_1, \ldots, Ec'_k$, $Ec_2, \ldots, Ec_n$ zero-ary entity composers; thus *a view of the structure* of e is given by the following graph

Notice that we have spoken of *a* view of the structure of e and not of *the* view of the structure, since there may be different views of the structure of an entity; think for example of a system where the subcomponents which reach an error situation do not affect any more the activity of the system, thus we can have, e.g.,

$$e = Ec(e_1, \ldots, e_n) = Ec''(e_1, \ldots, e_n, e_{n+1})$$

where e_{n+1} corresponds to an error situation and so there are two different views of the structure of e.

Clearly, in general, the activity of a structured entity is determined by the activity of its subcomponents, see the examples in the following subsection.

Since we are interested in having an algebraic framework, for getting a specification formalism integrating the specifications of abstract data types, we precisely define the models, whose main features have been introduced above, as algebras; see the Appendix A for a summary of algebraic definitions and notations.

Entity signatures should provide syntactic elements for representing all parts of the models, so they are particular signatures with predicates having:

- sorts for the various types of entities (*entity sorts*),
- predicate symbols corresponding to the transition relations of the entities and sorts for the associated labels,
- operation symbols corresponding to the entity composers;

obviously also sorts, operation and predicate symbols for representing and manipulating the data handled by the entities. Entity algebras will be partial algebras with predicates on entity signatures s.t. we can find at least a view of the structure of each element of entity sort. Entity signatures and algebras are formally given below.

- An *entity signature* $E\Sigma$ is a pair (Σ, ES), where $\Sigma = (S, OP, PR)$ is a signature with predicates, $ES \subseteq S$ (the set of the entity sorts) s.t. for each $es \in ES$ there exists a sort $l\text{-}es \in S$ (labels of the transitions of entities of sort es) and a predicate symbol $\_ \xrightarrow{\_} \_ : es \times l\text{-}es \times es \in PR$ (representing the transitions of entities of sort es) .

  For a given $E\Sigma$, the family of the *entity composers* is

  $$EC(E\Sigma) = \{EC_{w,es}\}_{w \in ES^*, es \in ES},$$

  where for all w, es $EC_{w,es} = OP_{w,es}$.

– An *entity algebra* EA on $E\Sigma = (\Sigma, ES)$ is just a Σ-many-sorted partial algebra with predicates s.t. each $e \in \mathrm{EA}_{es}$, with $es \in ES$, is representable by a composition of interpretations of entity composer (more precisely, it is the interpretation of a term built using only the entity composer operations).
– The *structure views* on EA are the ordered trees with the nodes labelled by entity composers s.t. a node labelled by $Ec: s_1 \times \ldots \times s_n \to s$, has exactly n sons whose roots are labelled by composers with result sorts $s_1, \ldots, s_n$ respectively. Now we define by induction when one of such trees is a *view of the structure* of an entity:
  * Ec is a view of the structure of e iff $e = Ec$.
  * is a view of the structure of e iff there exist $e_1, \ldots, e_n$ s.t. $e = Ec(e_1, \ldots, e_n)$ and $ev_1, \ldots, ev_n$ are views of the structures of $e_1, \ldots, e_n$ respectively.
– We say that e is a *(proper) subentity* of a view ev iff there exists a (proper) subtree of ev which is a view for e.

2.2 A Small Example

For illustrating the models introduced before we give an entity algebra representing the reactive system modelling the executions of the programs of a very simple concurrent language CL.

In CL programs whatever number of sequential processes perform commands, whose syntax is given below, and evolve in an interleaving way interacting among them by exchanging signals in a synchronous way (handshaking communication).

$$com ::= \underline{\text{skip}} \mid com_1; com_2 \mid \underline{\text{send-signal}}(sig) \mid \underline{\text{rec-signal}}(sig) \mid com_1 + com_2 \mid seq$$

<u>skip</u> is the null command, <u>send-signal</u> and <u>rec-signal</u> are the commands for the signal exchange and "+" is the nondeterministic choice, where *sig* and *seq* are the nonterminals for signals and sequential commands not further detailed.

Let $C\Sigma$ be the entity signature given below, where "–" precedes comments, the key word "**esorts**" the list of the entity sorts, "**sorts**" the list of the remaining sorts and "**opns**", "**preds**" the list of the operations and predicate symbols respectively with their functionalities. Moreover for some of the operation and predicate symbols, we use a mixfix notation; for instance,

$$\_ \mid\mid \_ : proc \times proc \to proc$$

means that we shall write $p_1 \mid\mid p_2$ instead of $\mid\mid(p_1, p_2)$; i.e. terms of appropriate sorts replace underscores.

sig $C\Sigma$ =
esorts $prog, proc$
– in this case there are two kinds of entities:
– CL programs and processes and so two entity sorts

sorts *l-prog*, *l-proc*, *sig*
opns

$C\colon \to proc$ for all commands C defined by the above BNF

– entity composers for entities of sort *proc*
– (they are simple entities, i.e. without dynamic subcomponents)

$TAU\colon \to l\text{-}proc$ – process internal activity
$SEND, REC\colon sig \to l\text{-}proc$

– for labelling the process transitions corresponding to sending and
– receiving signals

$\alpha, \beta, \ldots\colon \to sig$ – the signals are just Greek letters
$\underbrace{\_ \,||\, \_ \ldots \_ \,||\, \_}_{n \text{ times}}\colon \underbrace{proc \times \ldots \times proc}_{n \text{ times}} \to prog \qquad n \geq 1$

– entity composers for programs

$\tau\colon \to l\text{-}prog$ – program internal activity

preds

$\_ \xrightarrow{\_} \_\colon proc \times l\text{-}proc \times proc$ – transition predicate for processes
$\_ \xRightarrow{\_} \_\colon prog \times l\text{-}prog \times prog$ – transition predicate for programs

Let CL be the term-generated $C\Sigma$-algebra s.t.:

- the carriers of sorts *sig*, *l-proc* and *l-prog* are the sets of the ground terms of the corresponding sorts on $C\Sigma$; the carrier of *proc* is the set of the quotient of the ground terms of sort *proc* w.r.t. the identifications requiring that ";" is associative, "+" is associative, commutative and both have "$\underline{\text{skip}}$" as identity; while CL_{prog} is the set of the finite non-empty parts of CL_{proc}.
  Here for simplicity the interpretation in CL of *Symb*, either a predicate or an operation symbol, will be simply written *Symb* and analogously for ground terms, thus t^{CL} will be written t.
- $p_1 \,||\, \ldots \,||\, p_n = \{p_1, \ldots, p_n\}$ for all $n \geq 1$, while the interpretations of the other operations are defined in the obvious way;
- the interpretations of the transition predicates are defined by the following inductive rules.

$$\frac{}{sq \xrightarrow{TAU} p'} \quad sq \text{ sequential command}$$

$$\frac{}{\underline{\text{send-signal}}(s) \xrightarrow{SEND(s)} \underline{\text{skip}}} \qquad \frac{}{\underline{\text{rec-signal}}(s) \xrightarrow{REC(s)} \underline{\text{skip}}}$$

$$\frac{p_1 \xrightarrow{l} p_1'}{p_1; p_2 \xrightarrow{l} p_1'; p_2} \qquad \frac{p_1 \xrightarrow{l} p_1'}{p_1 + p_2 \xrightarrow{l} p_1'}$$

$$\frac{p_1 \xrightarrow{TAU} p_1'}{p_1 \,||\, p_2 \,||\, \ldots \,||\, p_n \xRightarrow{\tau} p_1' \,||\, p_2 \,||\, \ldots \,||\, p_n}$$

$$\frac{p_1 \xrightarrow{SEND(s)} p_1' \quad p_2 \xrightarrow{REC(s)} p_2'}{p_1 \,||\, p_2 \,||\, p_3 \,||\, \ldots \,||\, p_n \xRightarrow{\tau} p_1' \,||\, p_2' \,||\, p_3 \,||\, \ldots \,||\, p_n} \quad n \geq 2$$

Notice that since the interpretations of $+$ and $||$ are commutative, the above rules fully describe the activity of the CL programs.

The following examples show that our definition of entity algebras allows to formally describe several interesting situations occurring in reactive systems.

Different ways of composing some entities may be equivalent.

$$pg = \underline{\text{send-signal}}(\alpha) \mid\mid \underline{\text{rec-signal}}(\beta) \in \mathrm{CL}_{prog}$$

is an entity whose structure may be seen in two different ways; indeed pg is also equal to

$$\underline{\text{rec-signal}}(\beta) \mid\mid \underline{\text{send-signal}}(\alpha);$$

that means that in CL programs the order of the processes in parallel is not relevant.

The two views of the structure of pg are graphically represented by

Compositions of different groups of entities may be equivalent. It is possible that an entity has some views of its structure with different number of subentities; indeed, for example,

$pg' = \underline{\text{send-signal}}(\alpha) = \underline{\text{send-signal}}(\alpha) \mid\mid \underline{\text{skip}} = \underline{\text{send-signal}}(\alpha) \mid\mid \underline{\text{skip}} \mid\mid \underline{\text{skip}} = \ldots$

Thus in the CL programs the processes which cannot perform any action ($\underline{\text{skip}}$) do not matter.

Various views of the structure of pg' are graphically represented by:

Sharing of subentities. Consider a language CL_1 differing from CL only for having a multilevel parallelism instead of a flat one; we just take a new signature $C\Sigma_1$ obtained from $C\Sigma$ by replacing the various operations "$||$" (entity composers for programs) with

$$\_ : proc \to prog \qquad \_ \mid\mid\mid \_ : prog \times prog \to prog,$$

and give a $C\Sigma_1$-entity algebra CL_1 in the same way of CL. In this case an entity of sort $prog$ has either one subentity of sort $proc$ or two subentities of the same sort $prog$.

$$pg'' = (p_1 \mid\mid\mid p_2) \mid\mid\mid (p_1 \mid\mid\mid p_3),$$

is an entity where the subentity represented by p_1 is shared between the subentities "$p_1 ||| p_2$" and "$p_1 ||| p_3$"; a view of its structure is graphically represented by

Entities may terminate and new entities may be created. Consider a language CL_2 differing from CL only for having commands corresponding to creation and termination of processes ($\underline{\text{terminate}}$, $\underline{\text{create}}(p)$). We take a new signature $C\Sigma_2$ obtained by adding to $C\Sigma$ operations corresponding to the new commands and

$$TERM\colon \to l\text{-}proc \qquad CREATE\colon proc \to l\text{-}proc;$$

and give the algebra CL_2 in the same way as CL; where the transitions due to the new commands are given by

$$\frac{}{\underline{\text{terminate}} \xrightarrow{TERM} \underline{\text{skip}}} \qquad \frac{}{\underline{\text{create}}(p) \xrightarrow{CREATE(p)} \underline{\text{skip}}}$$

$$\frac{p_1 \xrightarrow{TERM} p_1'}{p_1 \,||\, p_2 \,||\, \ldots \,||\, p_n \overset{\tau}{\Longrightarrow} p_2 \,||\, \ldots \,||\, p_n}$$

$$\frac{p_1 \xrightarrow{CREATE(p)} p_1'}{p_1 \,||\, p_2 \,||\, \ldots \,||\, p_n \overset{\tau}{\Longrightarrow} p \,||\, p_1' \,||\, p_2 \,||\, \ldots \,||\, p_n}$$

Graphically an example of a creation and of a termination of a process are shown by the transitions:

These last examples show also that an entity can modify its structure during a transition.

3 The Syntax of the Metalanguage

Now we look for an appropriate language for expressing abstract requirements on reactive systems formally described by entity algebras.

As a first attempt we could use first-order formulae on entity signatures (see Appendix A); with this language we can express some interesting properties. Consider the signature $C\Sigma$ introduced in Section 2.2; the axiom

$$\neg(\alpha = \beta)$$

requires that the signals (static elements) represented by α and β should be different; while

$$p_1 \mid\mid p_2 = p_2 \mid\mid p_1$$

requires that the order of the process components of a CL program does not matter; and

$$\nexists p', p'', s, s' \,.\, tp \xrightarrow{SEND(s)} p' \wedge tp \xrightarrow{REC(s')} p''$$

requires that the process represented by the term tp cannot both receive and send signals.

However first-order logic on entity algebras has several limits:

1. concerning static properties: it cannot express properties about the structure of the entities without introducing some entity composers (as for the parallel composer $\_ \mid\mid \_$ before);
2. concerning dynamic properties: it can only express properties about the local activity (immediate future/past) of entities, but not e.g. liveness properties.

For overcoming these limits we extend first-order logic with:

1. Special predicates "*AllSub*" for checking which are the subentities of an entity. $\_ \; AllSub \; \_ : es \times ent\text{-}set$, given a set of entities $eset$ and an entity e of sort es, returns true iff $eset$ is the set of all subentities (proper and not) of e w.r.t. some view of its structure. (Obviously we have also to introduce operations and predicates for handling set of entities.) We have found that such predicates are enough for expressing all properties of interest on the structure of entities (e.g. an entity is simple [has no subcomponents], there is an upper bound to the number of subcomponents, the activity of an entity is completely determined by the activity of its subcomponents and so on).

2. Classical temporal logic combinators similar to those of the branching time logic CTL (see [21]); briefly introduced below.
   Given an entity e in an entity algebra EA, a global view of its activity is given by the set of all its *execution paths*, i.e. maximal sequences of labelled transitions of the form:
$$e \xrightarrow{l_1}{}_{\mathrm{EA}} e_1 \xrightarrow{l_2}{}_{\mathrm{EA}} e_2 \ldots$$
   (clearly such sequences may be either finite or infinite); a sequence as above represents a possible behaviour of e. Thus a branching time-style property on the activity of e may be given saying either that all paths (there exists a path) for e satisfies some condition or there exists a path for e satisfying some condition.
   In our metalanguage we have the following combinators, where we assume that et is a term of entity sort:
   - $\triangle$ (for all paths) s.t. $\triangle(et, \pi)$ holds iff "for every execution path σ starting from the entity represented by et the path formula π holds on σ";
   - $\bigtriangledown$ (exists a path) s.t. $\bigtriangledown(et, \pi)$ holds iff "exists an execution path σ starting from the entity represented by et on which the path formula π holds".

   We have borrowed $\triangle$ and $\bigtriangledown$ from [21].
   For the path formulae we have the combinators
   - $\Box$ (always), $\Diamond$ (eventually), for safety and liveness properties;
   - $[\lambda x \,.\, \phi]$ which holds on a path σ whenever ϕ holds of the first state of σ;
   - $<\lambda x \,.\, \phi>$ which holds on σ whenever ϕ holds of the first label of σ (if it exits)
   - and the usual first-order combinators.

   Notice that, due to the combinator $<\ldots>$ our logic includes also the so called "edge formulae" see [11].
   Here, for lack of room we consider only such simple combinators; see [8] and [17] for other combinators, [8] presents also some examples motivating the introduction of $[\ldots]$ and $<\ldots>$. Notice that the choice of the metalanguage combinators for dynamic properties is in some sense orthogonal w.r.t. those for the structural properties.

Our metalanguage is then defined by putting together first-order logic with the formulae briefly introduced in 1. and 2. Formally, let $E\Sigma$ be an entity signature $((S, OP, PR), ES)$; then the axioms on $E\Sigma$ are a subclass of the dynamic formulas of [8] on a new signature $E\Sigma^{\mathrm{ST}}$, obtained by enriching $E\Sigma$ with:

- the predicates $\_\ \mathit{AllSub}\ \_ : \mathit{ent\text{-}set} \times \mathit{es}$ for all $\mathit{es} \in \mathit{ES}$,
- the sort *ent-set*, whose elements are the finite sets of elements of any entity sort,
- the usual operations and predicates on finite sets of entities: $\emptyset$, $\_ \cup \_$, $\{\_\}$, $\_ \subseteq \_$.

Given a sort assignment X on $E\Sigma^{\mathrm{ST}}$ the sets $DF_{E\Sigma}(X)$ of *dynamic formulae* and $P_{E\Sigma}(X, es)$ of *path formulae* of sort $es \in ES$ are inductively defined as follows (where t, t_1, ..., t_n denote terms of appropriate sort on $E\Sigma^{\mathrm{ST}}$, X and we assume that sorts are respected).
dynamic formulae

$Pr(t_1, \ldots, t_n) \in DF_{E\Sigma}(X)$	if	Pr is a predicate of $E\Sigma^{\mathrm{ST}}$
$t_1 = t_2 \in DF_{E\Sigma}(X)$		
$\neg\phi_1$, $\phi_1 => \phi_2 \in DF_{E\Sigma}(X)$	if	$\phi_1, \phi_2 \in DF_{E\Sigma}(X)$

$\forall x \,.\, \phi \in DF_{E\Sigma}(X)$	if	$\phi \in DF_{E\Sigma}(X)$, $x \in X$
$\triangle(t,\pi) \in DF_{E\Sigma}(X)$	if	t has sort es, $\pi \in P_{E\Sigma}(X, es)$, with $es \in ES$

path formulae

$[\lambda x \,.\, \phi] \in P_{E\Sigma}(X, es)$	if	$x \in X_{es}$, $\phi \in DF_{E\Sigma}(X)$
$<\lambda x \,.\, \phi> \in P_{E\Sigma}(X, es)$	if	$x \in X_{l\text{-}es}$, $\phi \in DF_{E\Sigma}(X)$
$\neg\pi_1, \pi_1 => \pi_2 \in P_{E\Sigma}(X, es)$	if	$\pi_1, \pi_2 \in P_{E\Sigma}(X, es)$
$\forall x \,.\, \pi \in P_{E\Sigma}(X, es)$	if	$\pi \in P_{E\Sigma}(X, es)$, $x \in X$
$\Box\pi \in P_{E\Sigma}(X, es)$	if	$\pi \in P_{E\Sigma}(X, es)$

Moreover we consider the following derived combinators: $\exists$, $\vee$, $\equiv$ defined as usual; $\Diamond\pi =_{\text{def}} \neg\Box\neg\pi$ and $\bigtriangledown(t,\pi) =_{\text{def}} \neg \triangle (t, \neg\pi)$.

4 Validity of Formulae and Semantics of a Specification

Since $\phi \in DF_{E\Sigma}(X)$ is built on the richer signature $E\Sigma^{\text{ST}}$, first of all the validity of ϕ in an $E\Sigma$-entity algebra EA is the validity of ϕ in EA^{ST}, an appropriate extension of EA to an $E\Sigma^{\text{ST}}$-algebra, where the added sorts, operations and predicates are interpreted in the obvious way (e.g.: $eset\ AllSub^{\text{EA}^{\text{ST}}}\ e$ holds iff there exists ev a view of the structure of e s.t. $eset$ is the set of all subentities of ev), see [4] for a complete definition of EA^{ST}.

Moreover we need some preliminary definitions. We denote by $PATH(\text{EA}, es)$ the set of the execution paths for the entities of sort es, i.e. the set of all sequences having either of the two forms below:

(1) $e_0\ l_0\ e_1\ l_1\ e_2\ l_2 \ldots e_n\ l_n \ldots$ (infinite path)
(2) $e_0\ l_0\ e_1\ l_1\ e_2\ l_2 \ldots e_n\ l_n \ldots e_k\ k \geq 0$ (finite path)
where for all $n \in \mathbb{N}$: $e_n \in \text{EA}_{es}$, $l_n \in \text{EA}_{l\text{-}es}$ and $(e_n, l_n, e_{n+1}) \in \to^{\text{EA}}$; moreover, in (2) for no e, l: $(e_k, l, e) \in \to^{\text{EA}}$ (there are no transitions starting from the final element of a finite path).

If σ is either (1) or (2) above, then

- $S(\sigma)$ denotes the first element of σ: e_0;
- $L(\sigma)$ denotes the second element of σ: l_0 (if it exists);
- $\sigma|_n$ denotes the path $e_n\ l_n\ e_{n+1}\ l_{n+1}\ e_{n+2}\ l_{n+2} \ldots$ (if it exists).

Let EA be an $E\Sigma$-entity algebra and $V\colon X \to \text{EA}$ be a variable evaluation; we define by multiple induction when a dynamic formula $\phi \in DF_{E\Sigma}(X)$ *holds in* EA *under* V (written $\text{EA}, V \models \phi$) and when a path formula $\pi \in P_{E\Sigma}(X, es)$ *holds on a path* $\sigma \in PATH(\text{EA}, es)$ *under* V (written $\text{EA}, \sigma, V \models \phi$).
dynamic formulae

$\text{EA}, V \models Pr(t_1, \ldots, t_n)$	iff	$(t_1^{\text{EA}^{\text{ST}},V}, \ldots, t_n^{\text{EA}^{\text{ST}},V}) \in Pr^{\text{EA}^{\text{ST}}}$
$\text{EA}, V \models t_1 = t_2$	iff	$t_1^{\text{EA}^{\text{ST}},V} = t_2^{\text{EA}^{\text{ST}},V}$ (both sides must be defined and equal)
$\text{EA}, V \models \neg\phi$	iff	$\text{EA}, V \not\models \phi$
$\text{EA}, V \models \phi_1 => \phi_2$	iff	either $\text{EA}, V \not\models \phi_1$ or $\text{EA}, V \models \phi_2$

$\mathrm{EA}, V \models \forall x \,.\, \phi$	iff	for all $v \in \mathrm{EA}_s^{\mathrm{ST}}$, with s sort of x, $\mathrm{EA}, V[v/x] \models \phi$
$\mathrm{EA}, V \models \triangle(et, \pi)$	iff	$et^{\mathrm{EA}^{\mathrm{ST}}, V}$ is defined and for all $\sigma \in \mathit{PATH}(\mathrm{EA}, es)$, with es sort of et, s.t. $S(\sigma) = et^{\mathrm{EA}^{\mathrm{ST}}, V}$, $\mathrm{EA}, \sigma, V \models \pi$

path formulae

$\mathrm{EA}, \sigma, V \models [\lambda x \,.\, \phi]$	iff	$\mathrm{EA}, V[S(\sigma)/x] \models \phi$
$\mathrm{EA}, \sigma, V \models <\lambda x \,.\, \phi>$	iff	either $\mathrm{EA}, V[L(\sigma)/x] \models \phi$ or $L(\sigma)$ is not defined
$\mathrm{EA}, \sigma, V \models \neg \pi$	iff	$\mathrm{EA}, \sigma, V \not\models \pi$
$\mathrm{EA}, \sigma, V \models \pi_1 => \pi_2$	iff	either $\mathrm{EA}, \sigma, V \models \pi_1$ or $\mathrm{EA}, \sigma, V \models \pi_2$
$\mathrm{EA}, \sigma, V \models \forall x \,.\, \pi$	iff	for all $v \in \mathrm{EA}_s$, with s sort of x, $\mathrm{EA}, \sigma, V[v/x] \models \pi$
$\mathrm{EA}, \sigma, V \models \Box \pi$	iff	for all $j \geq 0$ s.t. $\sigma\|_j$ is defined, $\mathrm{EA}, \sigma\|j, V \models \pi$.

A formula $\phi \in \mathit{DF}_{E\Sigma}(X)$ is *valid* in EA (written $\mathrm{EA} \models \phi$) iff $\mathrm{EA}, V \models \phi$ for all variable evaluations V.

A specification is a pair $(E\Sigma, Ax)$, where $Ax \subseteq \mathit{DF}_{E\Sigma}(X)$, and usually its semantics is the class of its models, i.e. of the $E\Sigma$-algebras satisfying all formulae in Ax. But, consider the specification

$$\mathit{TWO\_SIMPLE} = (S\Sigma, \{\theta\})$$

where $S\Sigma$ is the entity signature:

sig $S\Sigma$ =
  esorts *syst*, *proc*
  – there are entities of two kinds: systems and processes
  sorts *l-syst*, *l-proc*
  preds
    $\_ \xrightarrow{-} \_ : \mathit{proc} \times \mathit{l\text{-}proc} \times \mathit{proc}$
    $\_ \stackrel{-}{\Longrightarrow} \_ : \mathit{syst} \times \mathit{l\text{-}syst} \times \mathit{syst}$

and θ the following formula of our metalanguage

$$\mathit{eset}\ \mathit{AllSub}\ e => \exists p_1, p_2 : \mathit{proc} \,.\, p_1 \neq p_2 \wedge p_1 \neq e \wedge p_2 \neq e \wedge \mathit{eset} = \{p_1, p_2, e\}$$

which formalizes the requirement "the entities of sort *syst* are the parallel composition of two simple (i.e., without internal parallelism) entities of sort *proc*" (recall that an entity is always a subentity of itself). The models of *TWO_SIMPLE* are the $S\Sigma$-entity algebras EA s.t. θ holds in EA; but no such algebras exist. Indeed, since there are no operations of sort *syst* in $S\Sigma$, there are no entity composers and so, for each $S\Sigma$-entity algebra EA

$$\mathrm{EA} \models \mathit{eset}\ \mathit{AllSub}\ e \quad \text{iff} \quad \mathit{eset} = \{e\}.$$

Thus the specification *TWO_SIMPLE* has no models. However it is easy to exhibit various entity algebras describing concurrent systems with two and only two simple process subcomponents; but they are entity algebras on signatures richer than $S\Sigma$. For example, all $P\Sigma$-entity algebras, where $P\Sigma$ is the entity signature $S\Sigma$ enriched by the operations

$Nil\colon \to proc$
$\_ \cdot \_\colon l\text{-}proc \times proc \to proc$
$\_ + \_\colon proc \times proc \to proc$
$\tau\colon \to l\text{-}proc$
$\_||\_\colon proc \times proc \to system$

seem sensible models of TWO_SIMPLE.

Our solution is to take as models of a specification $(E\Sigma, Ax)$ the entity algebras on a signature $E\Sigma'$, extending $E\Sigma$, satisfying the axioms in Ax; the extra syntactic elements in $E\Sigma' - E\Sigma$ (entity composers) allow us to describe the structure of the entities. Then the class of the models of an entity specification $(E\Sigma, Ax)$ is

$\{\text{EA} | \text{EA}$ is an $E\Sigma'$-entity algebra, where $E\Sigma \subseteq E\Sigma'$ and for all $\phi \in Ax$ $\text{EA} \models \phi\ \}$.

5 Examples and Applications

5.1 Requirements Specification of a Net of Workstations

Assume that we need to specify the initial requirements for a net of workstations; for simplicity we list only some of them, choosing the more interesting.

The net consists of several workstations and on each workstation several processes may run in parallel; moreover processes may be moved from a workstation to another. Some relevant properties of the net are informally listed below.

P1) Each workstation is deadlock free, i.e. if it is unable to perform any activity, then also each process component is so.
P2) Each workstation has only simple subcomponents (i.e., without internal parallelism).
P3) If a workstation receives a process from some other one, then such process should never reach some error situation, never create other processes and cannot go on forever to perform some activity.
P4) The net includes either a workstation with a cartridge reader or a workstation with a CD player but not both.
P5) The processes on a workstation perform their activity in an interleaving way (i.e. it cannot happen that two processes perform some activity simultaneously).

Now we formalize the above requirements in a very abstract way using our metalanguage, i.e. formalizing exactly only the above properties and thus without any kind of overspecification; e.g., we do not fix the topology of the net, nor the architecture of the workstations, nor the policy followed by the process scheduler, nor the commands executed by processes and so on, since these features are not a consequence of the informal requirements.

Note that in several usual specification formalisms, properties about the dynamic activity of the workstations, as P1) and P3), cannot be expressed also making some overspecification, since it is not possible to speak of the dynamic components of a systems and of their dynamic properties

Below the lines preceded by $-$ are line-by-line comments of the axioms, where the word in boldface directly corresponds to some logical combinators, and we use the following abbreviation for the formula checking whether e' is a proper subcomponent of e.

$$e'\ Is\_Sub\ e =_{\text{def}} \exists eset\,.\, eset\ AllSub\ e\ \wedge\ e' \in eset\ \wedge\ e' \neq e$$

spec A_NET =

esorts $net, workstat, proc$ – there are three kinds of entities

sorts $l\text{-}net, l\text{-}workstat, l\text{-}proc$

opns

$CREATE\colon proc \to l\text{-}proc$

– labels for the transitions corresponding to create new processes

$REC\colon proc \to l\text{-}workstat$

– labels for the transitions corresponding to receive a process from another

– workstation

preds

$Error\colon proc$ – determines the process error situations

$Has\_Cartridge, Has\_CD\colon workstat$

– determines the workstations having a cartridge reader and a CD player

– respectively

$\_ \stackrel{-}{\longrightarrow} \_\colon proc \times l\text{-}proc \times proc$

$\_ \stackrel{-}{\Longrightarrow} \_\colon workstat \times l\text{-}workstat \times workstat$

$\_ \stackrel{-}{\sim\!\sim\!\sim>} \_\colon net \times l\text{-}net \times net$

axioms

– P1)

$\nexists w', l \,.\, w \stackrel{l}{\Longrightarrow} w' \wedge$

– **if** w is unable to perform any activity **and**

$p \; Is\_Sub \; w =>$

– p is one of its proper subcomponents **then**

$\nexists p', l_1 \,.\, p \stackrel{l_l}{\longrightarrow} p'$

– p is unable to perform any activity

–

– P2)

$p \; Is\_Sub \; w => \nexists e \,.\, e \; Is\_Sub \; p$

– **if** p is a proper subcomponent of w **then** p has not proper subcomponents

–

– P3)

$w \stackrel{REC(p)}{\Longrightarrow} w' =>$

– **if** p is received by w, **then**

$\triangle(p, \Box[\lambda x \,.\, \neg Error(x)]) \wedge$

– **in each case** p will never reach an error situation **and**

$\triangle(p, \Box<\lambda l \,.\; \nexists p' \,.\, l = CREATE(p')>) \wedge$

– **in each case** p will never create a new process **and**

$\triangle(p, \;\; [\lambda x \,.\; \nexists x', l \,.\, x \stackrel{l}{\longrightarrow} x'])$

– **in each case** p cannot go on for ever to act (i.e., eventually it will

– reach a state where it cannot perform any activity)

–

– P4)

$\exists w \,.\, w \; Is\_Sub \; n \;\wedge\; (Has\_Cartridge(w) \;\vee\; Has\_CD(w)) \wedge$

– a net n has a proper subcomponent w having either a cartridge reader or

– a CD player **and**

$\neg\exists w_1, w_2 \,.$

– has not both

$w_1 \; Is\_Sub \; n \;\wedge\; Has\_Cartridge(w_1) \;\wedge\; w_2 \; Is\_Sub \; n \;\wedge\; Has\_CD(w_2))$

– a subcomponent having a cartridge and one having a CD player

– P5)
$w \stackrel{l}{\Longrightarrow} w' =>$
– **if** w perform some activity **then**
    $\exists eset, p, p', l' \,.\, (esetAllSubw \;\wedge\; p \in eset \;\wedge\; p \neq w \;\wedge\; p \stackrel{l'}{\longrightarrow} p' \;\wedge$
  – it has a proper subcomponent p performing some local activity **and**
    $(es - \{p\}) \cup p' \; AllSub \; w')$
  – such subcomponents is the only one which has modified its state
  – during the transition (i.e. such activity consists of some activity of only
  – one of its subcomponents)

We have experimented the above line-by-line natural language comments in some industrial applications ([18, 19]) and fairly believe that this device is an essential ingredient for making formal specifications acceptable to a wide community of users.

5.2 An Industrial Case Study

Our metalanguage for expressing the requirements of reactive system has been used in two industrial case studies in a project in cooperation with ENEL - SPA (the Italian national board for electric power). The two cases concern respectively the specification of a hydro-electric central for the production of electricity and a high-voltage substation for the distribution of electricity handled by automatisms (see [18, 19]). Here we briefly try to sketch the specification of the first case enlightening the role of the metalanguage.

The high-voltage substation has been specified at three different levels of abstraction.

Level 1: It formalizes the most relevant properties of the substation; this specification could be used e.g. in a contract with a firm realizing the plant. The substation is made by "functional units" of several kinds (Ae, Dd and Fa) and of two metallic bars, Ae's are put on one bar, while Dd's and Fa's are conneted to both bars; below there is a graphical representation of the structure of a substation having 6 functional units.

In this case the dynamic subcomponents are the functional units and using our framwork we can describe how they are organized in a substation; the possibility of describing sharing of subcomponents (see section 2.2) allows to formalize the fact that Dd's and Fa's are connected to both bars.

The substation can receive orders of performing operations on the component functional units from an operator and informs such operator about the result of the required operations.

The abstract requirements specify what should happen when an order is received, but do not completely describe the execution of such orders. A sample property is:

$st \xrightarrow{Order(Close,id)} st' \wedge$

– **if** the substation receives the order of closing *id* **and**

$(Is\_Ae(st, id) \vee Is\_Dd(st, id)) \wedge Is\_Open(st, id) =>$

– *id* is either an Ae or a Dd that is open, **then**

$\triangle(st',$

– **in each case**

$(([\lambda x . Is\_Closed(x, id)] \wedge$

– **eventually** the functional unit *id* will become closed **and**

$<\lambda l . l = Performed\_Operation>) \vee$

– **eventually** it will inform the operator that the operation has

– been performed **or**

$[\lambda x . Failure\_In\_Station(x)]))$

– **eventually** there will be a failure in the substation

Level 2: Here the design of the substation is refined by introducing an automatism for handling the operation received from the operator. Here the abstract requirements are only about the dynamic activity of the automatism while there are no requirements on its structure; instead the structure of the plant is fully specified by completely describing an appropriate realization of the functional units using standard devices and also the interactions of the automatism with the devices are completely defined. Clearly this specification is an implementation of the first one: i.e. all of its models are models of the first.

Level 3: Here the design of the substation is completed by defining a particular automatism. This is not a requirement specification, but instead a design specification, i.e. a formal definition of one very specific reactive system, so we do not need to use all metalanguage; it is sufficient to use a small subset, precisley only the conditional axioms (i.e., formulae of the form $\wedge_{1 \geq i \geq nI} \phi_1 => \phi_{n+1}$, where for $i = 1, \ldots, n+1$ ϕ_i is either an equation or an atom of the form $Pr(t_1, \ldots, t_m)$) and to take as models of a specifications the initial ones, see e.g. [1, 2, 3, 5]. Thus we have a uniform framework where to give specifications of reactive systems at different levels of abstraction.

6 Conclusion

The essential novelty of what we have presented lies in the possibility of specifying within the same formalism requirements about the static structure and the dynamic activity of a system.

Compared to the many formalisms using various forms of temporal logics, we have two distinguished features: the possibility of dealing both with different entities (of different sorts) and with the subcomponents of an entity, without lowering the abstraction level of a specification; moreover our formalism includes the usual specifications of abstract data types and it allows also to give integrate specifications of the dynamic and of the static features of a system.

The formalism has a clean mathematical support in the definition of an appropriate institution; to this end a key role is played by the definition of the class of models, which are entity algebras over extended signatures.

There is no room here for illustrating the possibility of relating such abstract requirement specifications to the design level specifications (e.g. the SMoLCS specifications of [2]); this can be done following an algebraic approach based on a notion of implementation, due to Sannella-Wirsing [22] (see [8, 4] for some examples).

Finally it may be of interest to mention the fact that the approach presented here is currently being used in some industrial case studies for relating requirements to more concrete design specifications, which have been already given (see Section 5.2).

A Algebras with Predicates

A *signature with predicates* is a triple $\Sigma = (S, OP, PR)$, where

- S is a set (the set of the *sorts*);
- OP is a family of sets: $\{OP_{w,s}\}_{w \in S^*, s \in S}$[2]; $Op \in OP_{w,s}$ is an *operation symbol* (of arity w and result s);
- PR is a family of sets: $\{PR_w\}_{w \in S^*}$; $Pr \in PR_w$ is a *predicate symbol* (of arity w).

We write $Op\colon s_1 \times \ldots \times s_n \to s$ for $Op \in OP_{s_1 \ldots s_n, s}$ and $Pr\colon s_1 \times \ldots \times s_n$ for $Pr \in PR_{s_1 \ldots s_n}$. A *partial Σ-algebra with predicates* (shortly a Σ-algebra) is a triple

$$A = (\{A_s\}_{s \in S}, \{Op^A\})_{Op \in OP}, \{Pr^A\}_{Pr \in PR})$$

consisting of the carriers associated with the sorts, the interpretations of the operation symbols and the interpretations of the predicate symbols; i.e.:

- if $s \in S$, then A_s is a set;
- if $Op\colon s_1 \times \ldots \times s_n \to s$, then $Op^A\colon A_{s_1} \times \ldots \times A_{s_n} \to A_s$ is a partial function;
- if $Pr\colon s_1 \times \ldots \times s_n$, then $Pr^A \subseteq A_{s_1} \times \ldots \times A_{s_n}$.

Usually we write $Pr^A(a_1, \ldots, a_n)$ instead of $(a_1, \ldots, a_n) \in Pr^A$.

Given an S-indexed family of sets of variables X, the *term algebra* $T_\Sigma(X)$ is the Σ-algebra defined as follows:

- $x \in X_s$ implies $x \in T_\Sigma(X)_s$;
- $Op \in OP_{\Lambda,s}$ implies $Op \in T_\Sigma(X)_s$;
- $t_i \in T_\Sigma(X)_{s_i}$ for $i = 1, \ldots, n$ and $Op \in OP_{s_1 \ldots s_n, s}$ imply $Op(t_1, \ldots, t_n) \in T_\Sigma(X)_s$;
- $Op^{T_\Sigma(X)}(t_1, \ldots, t_n) = Op(t_1, \ldots, t_n)$ for all $Op \in OP$;
- $Pr^{T_\Sigma(X)} = \emptyset$ for all $Pr \in PR$.

If $X_s = \emptyset$ for all $s \in S$, then $T_\Sigma(X)$ is simply written T_Σ and its elements are called *ground terms*. If A is a Σ-algebra, $t \in T_\Sigma(X)$ and $V\colon X \to A$ is a *variable evaluation*, i.e. a sort-respecting assignment of values in A to all variables in X, then the *interpretation of t in A w.r.t. V*, denoted by $t^{A,V}$, is defined by induction as follows:

- $x^{A,V} = V(x)$;
- $Op^{A,V} = Op^A$;
- $Op(t_1, \ldots, t_n)^{A,V} = Op^A(t_1^{A,V}, \ldots, t_n^{A,V})$.

if t is a ground term, then we use the notation t^A. A Σ-algebra A is *term-generated* iff for all $s \in S$, for all $a \in A_s$ there exists $t \in (T_\Sigma)_s$ s.t. $a = t^A$.

The sets $F_\Sigma(X)$ of *first-order formulae* on Σ and X are inductively defined as follows (where $t_1, \ldots, t_n$ denote terms of appropriate sort and we assume that sorts are respected):

- $Pr(t_1, \ldots, t_n) \in F_\Sigma(X)$ if $Pr \in PR$
- $t_1 = t_2 \in F_\Sigma(X)$
- $\neg\phi_1, \phi_1 => \phi_2 \in F_\Sigma(X)$ if $\phi_1, \phi_2 \in F_\Sigma(X)$
- $\forall x \,.\, \phi \in F_\Sigma(X)$ if $\phi \in F_\Sigma(X)$, $x \in X$

Let A be a Σ-algebra and $V\colon X \to A$ be a variable evaluation we define by induction when a formula $\phi \in F_\Sigma(X)$ *holds in A under V* (written $A, V \models \phi$)

[2] Given a set X, X^* denotes the set of the strings (finite sequences) over X.

- $A, V \models Pr(t_1, \ldots, t_n)$ iff $(t_1^{A,V}, \ldots, t_n^{A,V}) \in Pr^A$
- $A, V \models t_1 = t_2$ iff $t_1^{A,V} = t_2^{A,V}$ (both sides must be defined and equal)
- $A, V \models \neg\phi$ iff $A, V \not\models \phi$
- $A, V \models \phi_1 => \phi_2$ iff either $A, V \not\models \phi_1$ or $A, V \models \phi_2$
- $A, V \models \forall x \,.\, \phi$ iff for all $v \in A_s$, with s sort of x, $A, V[v/x] \models \phi$

A formula $\phi \in F_\Sigma(X)$ is *valid* in A (written $A \models \phi$) iff $A, V \models \phi$ for all evaluations V.

References

1. E. Astesiano, G.F. Mascari, G. Reggio, and M. Wirsing. On the parameterized algebraic specification of concurrent systems. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Proc. TAPSOFT'85, Vol. 1*, number 185 in Lecture Notes in Computer Science, pages 342–358. Springer Verlag, Berlin, 1985.
2. E. Astesiano and G. Reggio. SMoLCS-driven concurrent calculi. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proc. TAPSOFT'87, Vol. 1*, number 249 in Lecture Notes in Computer Science, pages 169–201. Springer Verlag, Berlin, 1987.
3. E. Astesiano and G. Reggio. Algebraic specification of concurrency (invited lecture). In *Recent Trends in Data Type Specification*, number 655 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1992.
4. E. Astesiano and G. Reggio. Entity institutions: Frameworks for dynamic systems, 1992. in preparation.
5. E. Astesiano and G. Reggio. A structural approach to the formal modelization and specification of concurrent systems. Technical Report PDISI-92-01, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Italy, 1992.
6. R.M. Burstall and J.A. Goguen. Introducing institutions. In E. Clarke and D. Kozen, editors, *Logics of Programming Workshop*, number 164 in Lecture Notes in Computer Science, pages 221–255. Springer Verlag, Berlin, 1984.
7. M. Cerioli and G. Reggio. Institutions for very abstract specifications. Technical Report PDISI-92-14, Dipartimento di Informatica e Scienze dell'informazione - Università di Genova, Italy, 1992.
8. G. Costa and G. Reggio. Abstract dynamic data types: a temporal logic approach. In A. Tarlecki, editor, *Proc. MFCS'91*, number 520 in Lecture Notes in Computer Science, pages 103–112. Springer Verlag, Berlin, 1991.
9. J. Fiadeiro and T. Maibaum. Describing, structuring and implementing objects. In J.W. de Bakker, W. P. de Roever, and G. Rozemberg, editors, *Foundations of Object-Oriented Languages, Proc. REX School/Workshoop*, number 489 in Lecture Notes in Computer Science, pages 274–310. Springer Verlag, Berlin, 1991.
10. I.S.O. LOTOS – A formal description technique based on the temporal ordering of observational behaviour. IS 8807, International Organization for Standardization, 1989.
11. L. Lamport. Specifying concurrent program modules. *ACM TOPLAS*, (5), 1983.
12. S. Mauw and G.J. Veltink. An introduction to PSF$_d$. In J. Diaz and F. Orejas, editors, *Proc. TAPSOFT'89, Vol. 2*, number 352 in Lecture Notes in Computer Science, pages 272 – 285. Springer Verlag, Berlin, 1989.
13. R. Milner. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1980.
14. G. Plotkin. An operational semantics for CSP. In D. Bjorner, editor, *Proc. IFIP TC 2-Working conference: Formal description of programming concepts*, pages 199–223. North-Holland, Amsterdam, 1983.

15. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency*, number 224 in Lecture Notes in Computer Science, pages 510–584. Springer Verlag, Berlin, 1986.
16. G. Reggio. Entities: an istitution for dynamic systems. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification*, number 534 in Lecture Notes in Computer Science, pages 244–265. Springer Verlag, Berlin, 1991.
17. G. Reggio. Event logic for specifying abstract dynamic data types. In *Recent Trends in Data Type Specification*, number 655 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1992.
18. G. Reggio, D. Bertello, and E. Crivelli. Specification of a hydro-electric central. Technical Report PDISI-92-13, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1992.
19. G. Reggio, A. Morgavi, and V. Filippi. Specification of a high-voltage substation. Technical Report PDISI-92-12, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1992.
20. W. Reisig. *Petri nets: an introduction.* Number 4 in EATCS Monographs on Theoretical Computer Science. Springer Verlag, Berlin, 1985.
21. C. Stirling. Comparing linear and branching time temporal logics. In *Temporal logics of Specification*, number 398 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1989.
22. M. Wirsing. Algebraic specifications. In van Leeuwen Jan, editor, *Handbook of Theoret. Comput. Sci.*, volume B, pages 675–788. Elsevier, 1990.

This article was processed using the LaTeX macro package with LLNCS style