Noname Manuscript-Nr.
(will be inserted by hand later)

Labelled Transition Logic: An Outline Technical Report DISI-TR-96-20[*]

E. Astesiano and G. Reggio

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Via Dodecaneso, 35 – Genova 16146 – Italy
Fax ++ 39 - 10 - 3536699
{ astes,reggio } @ disi.unige.it

Summary. In the last ten years we have developed and experimented in a series of projects, including industry test cases, a method for the specification of reactive concurrent systems both at the requirement and at the design level. We present here in outline its main technical features, providing pointers to appropriate references for more detailed presentations of single aspects and of the associated tools and reference manuals.

The overall main feature of the method is its logical/algebraic character, since it extends to labelled transition systems the logical/algebraic specification method of abstract data types and processes are viewed as data within first-order structures called LT-structures. Some advantages of the approach are the full integration of the specification of static data and of dynamic elements, which includes by free higher-order concurrency, and the exploitation of well-explored classical techniques in many respects, including implementation and tools.

On the other hand the use of labelled transition systems for modelling processes, inspired by CCS, allows us to take advantage, with appropriate extensions, of some important concepts, like communication mechanisms and observational semantics, developed in the area of concurrency around CCS, CSP and related approaches.

1. Introduction

Since the beginning of the 70's the concept of abstract data type, along with its formalization by means of many-sorted first-order structures (algebras), has been quite influential in the development of modular and correct software. But soon it was realized that handling static structures was not enough, since most software systems are dynamic, in the sense that they deal with structures evolving in time

[*] Work partially supported HCM-EXPRESS and MURST 40%.

and frequently also including some form of parallelism, concurrency, reactivity and distribution. Thus (a little) later on and extensively during the 80's, the modelization and specification of concurrent systems has become one of the leading issues both in theory and practice of software development.

This paper addresses the issue of abstract specification of concurrent systems from a logical (algebraic) viewpoint, which adopts and extends the classical logical/algebraic method for static data types. Its aim is to outline a rather comprehensive approach, whose characteristic feature is to see processes and concurrent systems as dynamic data, i.e. as elements of special dynamic sorts in a first-order structure. As in CCS and other approaches, processes are modelled as labelled transition systems, i.e. nondeterministic automata with labelled transitions. However, to keep the specification at the abstract level appropriate of the system to be specified, both states and labels are in general data of a suitable sort.

In order to support this view we adopt as basic mathematical structure what we call "LT-structure", i.e. a many-sorted first-order structure where some (dynamic) sorts, representing processes (states), have both an associate label sort and a ternary transition predicate, and so correspond to labelled transition systems.

Notice that an element representing a state in an LT-structure, because of the associate semantic equivalence, also represents the value of a whole process, the one with that initial state. Of course syntactically a term (expression) over some signature will represent such an element; thus our approach belongs to the family roughly driven by the "state-as-term" idea (CCS, CSP, and so on). A much different view also of interest and receiving some attention is the one driven by the "state-as-algebra" idea (see [23, 39] also for references).

Thus, we can use first-order logic over LT-structures to specify the properties on the activity of processes, since such logic includes as atomic formulae the assertions stating that a process may perform a certain labelled transition; this is the reason of the name of our formalism "LTL" (Labelled Transition Logic), i.e. a logic apt to "speak" about labelled transitions, with the notable characteristic to be just a particular first-order logic.

Because of the view that processes are data, that LT-structures are just particular first-order structures and that LTL is just a first-order logic, we can fully exploit the techniques of logical/algebraic specifications, with several advantages, which constitute distinguishing features of our approach.

- *Static and dynamic properties of processes*
  The formulae of LTL express both "*static*" and "*dynamic*" properties of processes. "Static" means properties about the static structure of the processes; e.g. the axiom "$b_1 + b_2 = b_2 + b_1$" requires that the nondeterministic choice operator $+$ of CCS is commutative. "Dynamic" means properties about the activity of the processes (properties about the transition predicate); e.g. the axiom "$l\ .\ b \xrightarrow{l} b$" requires that a CCS behaviour $l\ .\ b$ should be able to become b performing a transition labelled by l.
  Notice that static properties help also to express the dynamic ones in a simpler mode; e.g. the unique axiom "$b_1 \xrightarrow{l} b_1' \supset b_1 + b_2 \xrightarrow{l} b_1'$" completely characterizes the CCS nondeterministic choice $+$, since $+$ is commutative (by the static properties).

- *Integration of classical abstract data type specifications and process specifications*
  Since the LTL formulae include those of the first-order logic with equality, they can also be used to express the properties of the data used by the processes (e.g., values, messages, ...); LT-specifications allow us to integrate usual specifications of abstract data types with axiomatic specifications of processes. Moreover, since processes themselves are data, we can describe data types with process components (e.g., functions from and/or into processes), systems with different "types" of processes and complex systems where processes of some type are composed out of processes of other types.
- *Extension to concurrency of techniques/results of algebraic specification of abstract data types*
  We can extend all well-known results, techniques and methodological principles developed in the field of abstract data type specification to LT-specifications (and so to the specification of concurrency). As an outstanding example, we can model the now well established distinction between requirement and design specifications, with the associated notion of implementation, following the formalization given in the context of abstract data type specification, as loose and initial specifications, with a very general notion of implementation.
  Roughly speaking a *requirement specification* defines general requirements of a process/concurrent system for some level of abstraction; a bit more precisely it requires a process/system to satisfy at least some properties, and of course all their logical consequences; thus if SP is a requirement specification, then *Mod*(SP), the models of SP, consists of a variety of usually non-equivalent models, i.e. models differing even for some substantial aspect.
  A *design specification* is meant to describe exactly one process/concurrent system, for some level of abstraction; the idea is to determine a process/system as the one which satisfies "all and only" the properties listed in the specification and their logical consequences. Here an important constraint concerns the fact that the properties in a design specification have to be formulated in a way guaranteeing the uniqueness, apart from noninfluencial differences, of the system. More formally SP has to be such that *Mod*(SP) contains only models considered equivalent.
  By *implementation* we mean the combination of refinement and realization (reification), following the implementation notion developed by Wirsing and Sannella (see e.g. [64]): SP is implemented via α by SP$'$, where α is a function transforming specifications iff $\mathit{Mod}(\alpha(\mathrm{SP}')) \subseteq \mathit{Mod}(\mathrm{SP})$.

However in dealing with concurrent systems there are at least two aspects whose treatment goes beyond the traditional techniques of logical/algebraic specifications. The first concerns the semantics of a process, for which various notions have been proposed, depending on the observations of interest; in general those semantics cannot be appropriately characterized by standard techniques about initial, final or classical observational semantics. Some generalized notion of bisimulation and testing or equivalently the introduction of infinitary modal formulae is needed (see [2, 3, 4]).

The second has to do with the inadequacy of first-order logic for expressing requirements about system behaviour; in general a logic with modalities is needed, that can take the form of a temporal logic (see Sect. 5).

The paper gives a general outline of the fundamental aspects of an approach that has been developed, in its different aspects, since 1984 and has now reached a rather mature stage, also benefiting of the experience gained in some significant projects and industrial test cases, as reported in the conclusions. The approach consists by now of theoretical foundations, the formalism LTL, and the companion methodological software engineering guidelines with reference and user manuals, a supporting language and some basic automatic tools; this second part is usually known under the acronym SMoLCS. Here we will only deal with the basic underlying technical ideas, pointing to the literature for deeper technical developments, methodological issues and examples of applications, within a variety of languages.

In Sect. 2 we motivate and present the models, i.e. labelled transition systems and LT-structures; in Sect. 3 we deal with the design specifications, introducing conditional LT-specifications and their semantics. In Sect. 4 an extensive example is presented for illustrating the flexibility and modularity of the method, both for accommodating different kinds of parallelism and interactions and for leaving the door open, via parameterization, to further refinements. For a variety of more relevant applications of LTL/SMoLCS, see:

- specifications of case studies, as in [7, 10, 20, 60, 61];
- its usage as a base for formally defining other formalisms, e.g. [25], for formalizing in a modular way various kinds of higher-order algebraic Petri nets, [32] for a variant of extended shared Prolog, [5] for object-oriented features;
- and for defining the semantics of concurrent/parallel programming languages, as Ada, [1, 9, 12, 14].

The role and location of requirement specifications in our framework is briefly illustrated in Sect. 5, with basic concepts and pointers to appropriate references, and then Sect. 6 shows how the classical notion of implementation may relate different development steps. Finally in Sect. 7 we discuss the relationship to other work, and to compare them with LTL/SMoLCS we give some hints on how to use such other methods to specify the main example in Sect. 4. Here we only want to point out that LTL/SMoLCS has taken inspiration especially from three sources: SOS [55] for the use of labelled transition systems and their operational specifications, CCS [49, 50] for the concept of process as labelled transition tree with various observational semantics, and finally the Munich algebraic approach (see e.g. [28]) for a first example of an algebraic version of CCS.

The basic notations, definitions and results about the first-order specifications used in the paper are in Appendix A.

2. Models of dynamic elements

In this paper we use the words "dynamic element" to denote whatever kind of "thing" evolving along the time, without making any assumption about other aspects of its behaviour; thus here dynamic elements may be communicating/nondeterministic/... processes, reactive/parallel/concurrent/distributed ... systems, software/hardware processes/architectures, but also object-oriented systems (communities of interacting objects). Moreover, "concurrent dynamic element" denotes a dynamic element having several subcomponents, which are themselves dynamic elements.

2.1. Labelled transition systems for modelling dynamic elements

For modelling dynamic elements we adopt the well-known and accepted technique which consists in viewing a dynamic element as a labelled transition tree defined by a labelled transition system (see [49] and [56]).

Definition 1. *A* labelled transition system *(shortly* lts*) is a triple* $(STATE, LABEL, \rightarrow)$*, where* $STATE$ *and* $LABEL$ *are sets, whose elements represent respectively the* states *and the* labels *of the system, and* $\rightarrow\ \subseteq STATE \times LABEL \times STATE$ *is the* transition relation*. A triple* $(s, l, s') \in \rightarrow$ *is said a* transition *and it is usually denoted by* $s \xrightarrow{l} s'$*.* □

A dynamic element D may be represented by an lts $(STATE, LABEL, \rightarrow)$ and an initial state $s_0 \in STATE$; then the states in $STATE$ reachable from s_0 represent the intermediate (interesting) states of the life of D and the transition relation $\rightarrow$ the possibilities of D of passing from a state to another one. It is important to note that here a transition $s \xrightarrow{l} s'$ has the following meaning: D in the state s has the *capability* of passing into the state s' by performing a transition whose interaction with the external (to D) world is represented by the label l; thus l contains information on the conditions on the external world for the capability to become effective, and information on the transformation of such world induced by the execution of the action.

Given an lts we can associate with each state the so called *transition tree.* Precisely, a transition tree is a tree whose nodes are labelled by states and whose arcs are decorated by labels. Moreover, in a transition tree the order of the branches is not considered, two identically decorated subtrees with the same root are considered as a unique one, and there is an arc decorated by l between two nodes decorated respectively by s and s' iff $s \xrightarrow{l} s'$.

For example, let us consider the processes modelling the components of CA (a concurrent architecture, see Sect. 4.1). Such processes may be modelled by an lts whose states are pairs of the form $<c, lm>$, where c is a command and lm a local memory state, and whose labels include τ (in this paper, following the convention adopted by Milner for CCS, we use "τ" to label internal transitions) and $SEND(v, ch)$, $REC(v, ch)$ (sending and receiving a value v through a channel ch by handshaking communication).

A process with command part "$x := 0; Send(x, ch_0)$" and empty local memory is modelled by the transition tree reported in Fig. 1; the first transition labelled by τ is internal, while the second is a capability which will become effective only when the process is put in parallel with another one ready to receive the value 0 on channel ch_0 ($Empty[0/x]$ denotes the local memory state where 0 is associated with x).

The tree in Fig. 2 represents a process performing an input command on channel ch_0; it has infinite capabilities, since it can receive any possible value; the transition labelled by $REC(\overline{n}, ch_0)$ will become effective only when the process is in parallel with another one ready to send the value $\overline{n}$ on channel ch_0. The tree in Fig. 3 represents a process performing the nondeterministic choice between two output commands.

Also concurrent dynamic elements, i.e. structured dynamic elements with subcomponents which in turn are dynamic elements (examples of concurrent

$$\begin{tikzcd}
<x := 0; Send(x, ch_0), Empty> \arrow[d, "\tau"] \\
<Send(x, ch_0), Empty[0/x]> \arrow[d, "SEND(0, ch_0)"] \\
<Skip, Empty[0/x]>
\end{tikzcd}$$

Fig. 1. Transition tree associated with a CA process performing an output command

$$\begin{tikzcd}
 & <Rec(x, ch_0), Empty> \arrow[dl, "REC(0, ch_0)"'] \arrow[dr, "REC(1, ch_0)"] & & \cdots \\
<Skip, Empty[0/x]> & & <Skip, Empty[1/x]> & & \cdots
\end{tikzcd}$$

Fig. 2. Transition tree associated with a CA process performing an input command

$$\begin{tikzcd}
 & <Send(2, ch_1) + Send(3, ch_2), Empty> \arrow[dl, "SEND(2, ch_1)"'] \arrow[dr, "SEND(3, ch_2)"] & \\
<Skip, Empty> & & <Skip, Empty>
\end{tikzcd}$$

Fig. 3. Transition tree associated with a CA process performing a nondeterministic choice

dynamic elements are, for example, the CA architecture, see Sect. 4.1, and the Ada programs, which have as components the tasks, see [9]) may be modelled by using special lts's built by putting together (several) other lts's, whose transitions describe the activity of the component dynamic elements. So a dynamic element has dynamic components, said in the following *active components* (for example the CA processes and the task components of Ada programs); but, in general, it has also *passive components* (think for example of the buffer of CA architecture and of the shared memory of Ada programs). Here "passive" means that such components may change their states only as result of some transition of some active component.

For example, let us consider the CA architecture (see Sect. 4.1). It is a concurrent dynamic element and it could be modelled by an lts whose states have the form $p_1 \mid \ldots \mid p_n \mid bf$, where $p_1 \mid \ldots \mid p_n$ is a multiset of states of the lts modelling the CA processes (active components), bf is a value representing the state of the shared buffer (passive component), and whose transitions are determined by the transitions of the process components and by the buffer state. For instance, if $p_1 \xrightarrow{SEND(\overline{n},\overline{ch})} p'_1$ and $p_2 \xrightarrow{REC(\overline{n},\overline{ch})} p'_2$, then $p_1 \mid p_2 \mid pms \mid bf \xrightarrow{\tau} p'_1 \mid p'_2 \mid pms \mid bf$, where pms stands for a multiset of process states, those not taking part in the transition (handshaking communication between two processes); and if $p \xrightarrow{READ(\overline{n})} p'$ and $\overline{n}$ is the first value available in a buffer whose actual content is bf, then $p \mid pms \mid bf \xrightarrow{\tau} p' \mid pms \mid bf'$, where pms is as above and bf' is the buffer where $\overline{n}$ has been dropped from.

By associating with a dynamic element D the transition tree having root D we give an operational semantics: two dynamic elements are operationally equivalent whenever the associated transition trees are the same, see [49]. However in most cases such semantics is too fine, since it takes into account all details of the dynamic element activity. It may happen that two dynamic elements which we consider semantically equivalent have associated different transition trees. A simple case is when we consider the trees associated with two sequential CA processes (i.e., having only sequential commands) represented by two states p and p': they perform only internal activities (i.e., no interactions with the external world) and thus the associated transition trees (reported in Fig. 4) are unary trees, with all arcs labelled by τ. If we consider an input-output semantics,

$$\begin{tikzcd} p \arrow[r, "\tau"] & p_1 \arrow[r, "\tau"] & \ldots \arrow[r, "\tau"] & p_n \arrow[r, "\tau"] & p_{\mathrm{F}} \end{tikzcd}$$

$$\begin{tikzcd} p' \arrow[r, "\tau"] & p'_1 \arrow[r, "\tau"] & \ldots \arrow[r, "\tau"] & p'_n \arrow[r, "\tau"] & p'_{n+1} \arrow[r, "\tau"] & \ldots \arrow[r, "\tau"] & p'_m \arrow[r, "\tau"] & p'_{\mathrm{F}} \end{tikzcd}$$

Fig. 4. Transition trees associated with two sequential CA processes

then the two processes are equivalent iff p, p' are equivalent w.r.t. the input and p_F, p'_F are equivalent w.r.t. the output; the differences concerning other aspects (intermediate states, number of the intermediate transitions, etc.) are not considered.

From this simple example, we understand also that we can get various interesting semantics on dynamic elements modelled by lts's depending on what we observe (see e.g. [49, 53]). For instance, consider the well-known strong bisimulation semantics of Park [54] and Milner [49] and the trace semantics [43]. In the first case, two dynamic elements are equivalent iff they have the same associated transition trees after the states have been forgotten. In the second case, two dynamic elements are equivalent iff the corresponding sets of traces (streams of labels), obtained travelling along the maximal paths of the associated transition trees, are the same. In general, the semantics of dynamic elements depends on what we are interested to observe: i.e., the semantics of dynamic elements is observational, in a loose sense for the moment.

2.2. LT-structures

An lts can be represented by a (many-sorted) first-order structure A, (a concrete data type) with a signature having two sorts, say *state* and *label* (whose elements correspond to the states and the labels of the system), and a predicate

$$\_ \xrightarrow{\_} \_ : state \times label \times state,$$

representing the transition relation (see [28, 6]). The triple $(\mathrm{A}_{state}, \mathrm{A}_{label}, \rightarrow^{\mathrm{A}})$ is the corresponding lts. Clearly, we can handle in this way also lts's modelling concurrent dynamic elements; i.e. lts's where components of the states are in turn states of other lts's; in these cases we have first-order structures with several sorts corresponding to states and labels, together with the associated transition predicates. The first-order structures representing lts's are called "LT-structures" and are formally defined below.

Definition 2. *An* LT-signature *is a pair* (Σ, DS) *where:*

- $\Sigma = (S, OP, PR)$ *is a (many-sorted) first-order signature (see Appendix A.1),*
- $DS \subseteq S$ *(the elements in* DS *are the* dynamic sorts*, i.e. the sorts corresponding to states of lts's),*
- *for all* $ds \in DS$ *there exist a sort* $l\text{-}ds \in S - DS$ *(the sort of the labels) and a predicate* $\_ \xrightarrow{\_} \_ : ds \times l\text{-}ds \times ds \in PR$ *(the transition predicate).*

Given two LT-signatures, say $LT\Sigma = (\Sigma, DS)$ *and* $LT\Sigma' = (\Sigma', DS')$*, a* LT-signature morphism $\sigma: LT\Sigma \rightarrow LT\Sigma'$ *is a morphism of first-order signatures* $\sigma: \Sigma \rightarrow \Sigma'$ *s.t. for all* $ds \in DS$ $\sigma(ds) \in DS'$, $\sigma(l\text{-}ds) = l\text{-}\sigma(ds)$ *and* $\sigma(\_ \xrightarrow{\_} \_ : ds \times l\text{-}ds \times ds) = \_ \xrightarrow{\_} \_ : \sigma(ds) \times l\text{-}\sigma(ds) \times \sigma(ds)$ *(i.e. dynamic sorts with the related label sorts and transition predicates are preserved).* □

It is easy to see that LT-signatures and their morphisms form a category.

Definition 3. *An* LT-structure *on* $LT\Sigma = (\Sigma, DS)$ *(shortly* $LT\Sigma$*-structure) is a* Σ*-first-order structure.* □

Since every part of an lts is given as a data structure, we have an important difference w.r.t. the classical use of lts's, where states and labels are elements of some sets. Here states and labels are just elements of particular sorts in a first-order structure (i.e., particular data types) and so also the dynamic elements themselves are a data type (each dynamic element is given as its initial state); thus we can define dynamic elements which can be exchanged as values by other dynamic elements, dynamic elements and data having dynamic elements as subcomponents (e.g., functions from some data into dynamic elements) may be stored in memories, and so on. Hence the dynamic elements we model by LT-structures are potentially "higher-order", in a sense first introduced by the authors in [13] and now well-known because of underlying famous calculi like π-calculus [51, 52] and many others.

Many important applications of the higher-order paradigm have been shown; for example this approach was used to model Ada tasking (see [9]): the denotation of a task type is a function associating with some parameters the dynamic element corresponding to the task activity; the action of creating a new task type takes as parameter such function and as effect stores it as a value into the (shared) environment, and the creation of a new task of that type consists in adding to the system an instantiation of such function on appropriate parameters.

Since LT-structures are particular first-order structures, we take as homomorphisms the usual homomorphisms between such structures (see Appendix A.1) and show that they have good properties.

Definition 4. *Given two $LT\Sigma$-structures,* L *and* L', *an* LT-homomorphism h *from* L *into* L' *(written* $h\colon \mathrm{L} \to \mathrm{L}'$*) is a homomorphism from* L *into* L' *considered as Σ-first-order structures.* □

Since LT-homomorphisms preserve the truth of predicates, thus they preserve also the activity of the dynamic elements: if $h\colon \mathrm{L} \to \mathrm{L}'$ and $d \xrightarrow{l} d'$ in L, then $h(d) \xrightarrow{h(l)} h(d')$ in L'. In some sense the labelled transition tree associated with d is embedded into the corresponding one of $h(d)$.

Notice that given $d \in \mathrm{L}_{ds}$, with $ds \in DS$, an LT-homomorphism $h\colon \mathrm{L} \to \mathrm{L}'$ induces a total synchronous homomorphism of lts's (see e.g. [63]) between the lts associated with d and the one associated with $h(d)$.

It is easy to see that $LT\Sigma$-structures and LT-homomorphisms form a category.

Using the above LT-homomorphisms we can speak of initial elements in a class of $LT\Sigma$-structures and the following proposition shows their properties.

Proposition 1. *Let $\mathcal{L}$ be a class of $LT\Sigma$-structures,* $\mathrm{LI} \in \mathcal{L}$ *be initial in $\mathcal{L}$ and denote by $\models$ the usual notion of validity in first-order logic (see Appendix A.1 for a more detailed definition). Then:*

- $\mathrm{LI} \models t = t'$ *iff* ($\mathrm{L} \models t = t'$ *for all* $\mathrm{L} \in \mathcal{L}$ *);*
- *for all predicates Pr of $LT\Sigma$,*
  $\mathrm{LI} \models Pr(t_1, \dots, t_n)$ *iff* ($\mathrm{L} \models Pr(t_1, \dots, t_n)$ *for all* $\mathrm{L} \in \mathcal{L}$ *); thus in particular* $\mathrm{LI} \models td \xrightarrow{tl} td'$ *iff* ($\mathrm{L} \models td \xrightarrow{tl} td'$ *for all* $\mathrm{L} \in \mathcal{L}$*), i.e. in* LI *each dynamic element has in some sense the minimum amount of activity.*

Proof. From the properties of initial models in the category of many-sorted algebras with predicates (see Appendix A.1). □

Note: LT-structures introduced before may be regarded as *concrete dynamic-data types*, i.e. concrete data types of dynamic elements. Assume now that we want to abstract from the details of the concrete data, concerning the structure of (the states of) dynamic elements, of the static data and of the behaviour of the dynamic elements. By a procedure well-known in the case of usual data types (see [64] also for references), we can consider *abstract dynamic-data types* (shortly *ad-dt's*), where an ad-dt is an isomorphism class of LT-structures.

3. LT-specifications: design level

3.1. LT-specifications

In the previous section we have introduced LT-structures; here we tackle the problem of their specification.

Following the usual logical/algebraic style, we give a signature and then we describe the relationships holding among operations and predicates of that signature by means of axioms, obtaining what we call an *LT-specification*. In the past we have also used the names "algebraic transition systems" (e.g. in [6, 13]) and "dynamic specifications" (e.g. in [35, 59, 20]) for the same purpose.

Definition 5. *A pair $(LT\Sigma, AX)$, where $LT\Sigma = (\Sigma, DS)$ is an LT-signature and AX a set of first-order formulae on $LT\Sigma$ (i.e., on Σ), is called an* LT-specification*.* □

The purpose of an LT-specification is to define, by an appropriate semantics, LT-structures, or better isomorphism classes of LT-structures (abstract LT-structures).

As in the classic case of abstract data type specification, an LT-specification may determine different abstract LT-structures depending on the chosen semantics; for example we can consider the (isomorphism class of the) initial elements of the class of the models (initial approach) or the isomorphism class of a particular model satisfying some observational constraints. Analogously we can also consider LT-specifications with "loose" semantics: i.e., specifications which determine a class of abstract LT-structures.

LT-specifications determining one abstract LT-structure are called *design specifications*, since they may be used to define abstractly and formally the LT-structure describing the complete design of some dynamic elements; while LT-specifications with loose semantics are called *requirement specifications*, since they may be used to formally define the requirements on some dynamic elements by determining the class of all the abstract LT-structures describing dynamic elements satisfying those requirements.

3.2. A specification language for structured LT-specifications

In this subsection we introduce a (schema of) specification language for writing LT-specifications in a structured and modular way, which allows, if convenient, to reuse common subparts. A sufficiently powerful language can be obtained by considering five constructs only, along the pattern proposed by Wirsing in [64]. A difference with Wirsing's approach is that we follow a loose approach instead of the ultra-loose. Moreover, the dynamic features add an extra "dimension" to some of the operators, namely sum and export, as discussed below.

Here we assume a *loose semantics* for our specifications, therefore each language expression denotes a pair $(LT\Sigma, \mathcal{MOD})$, where $LT\Sigma$ is an LT-signature and $\mathcal{MOD}$ is a class of $LT\Sigma$-structures closed w.r.t. isomorphisms.

Let $LOOSE\_SP$ be the class of all pairs of the above form and SP_EXPR be the set of all language expressions (specification expressions) defined below; then the semantics of the language is given by a function

$$\mathbf{S}: SP\_EXPR \to LOOSE\_SP.$$

SP_EXPR and $\mathbf{S}$ are defined inductively by the rules marked with ($\bullet$) below; where for simplicity we do not distinguish between specifications and specification expressions, using SP, SP$'$, ... for both; if $\xi = (LT\Sigma, \mathcal{MOD})$, we write $Sig(\xi)$ and $Mod(\xi)$ for $LT\Sigma$ and $\mathcal{MOD}$ respectively; and union and containment of LT-signatures are taken componentwise (and w.r.t. the four components: S, OP, PR and DS).

Simple specifications Simple (or flat) specifications are the basic building blocks, thus:

($\bullet$) $(LT\Sigma, AX) \in SP\_EXPR$ for all LT-signatures $LT\Sigma$ and AX sets of first-order formulae on Σ;
$\mathbf{S}[(LT\Sigma, AX)] =$
$(LT\Sigma, \{\mathrm{L} \mid \mathrm{L}$ is an $LT\Sigma$-structure and $\mathrm{L} \models \theta$ for all $\theta \in AX$ $\})$.

In the examples we shall use the mixfix notation

sorts S
dsorts DS
opns OP
preds PR
axioms AX

for the simple LT-specification $(LT\Sigma, AX)$, where $LT\Sigma = (\Sigma, DS)$ and
$\Sigma = (S \cup DS \cup \{l\text{-}ds \mid ds \in DS\}, OP, PR \cup \{\ \longrightarrow : ds \times l\text{-}ds \times ds \mid ds \in DS\})$;
i.e. the canonical sorts and predicates are given implicitly.

Sum of specifications The sum is the basic construct for putting specifications together to build a larger one.

($\bullet$) $\mathrm{SP}_1 + \mathrm{SP}_2 \in SP\_EXPR$ for all $\mathrm{SP}_1, \mathrm{SP}_2 \in SP\_EXPR$;
$\mathbf{S}[\mathrm{SP}_1 + \mathrm{SP}_2] =$
$(LT\Sigma, \{\mathrm{L} \mid \mathrm{L}$ is an $LT\Sigma$-structure and $\mathrm{L}_{|LT\Sigma_i} \in Mod(\mathbf{S}[\mathrm{SP}_i])$ for $i = 1,2\})$, where $LT\Sigma = LT\Sigma_1 \cup LT\Sigma_2$ and $LT\Sigma_i = Sig(\mathbf{S}[\mathrm{SP}_i])$ for $i = 1,2$, (see Appendix A.1 for the definition of $\mathrm{L}_{|\ldots}$).

Notice that this construct allows us to specify static and dynamic features separately (and then combine them). More precisely, let for $i = 1$, 2 $LT\Sigma_i$ be the LT-signature $((S_i, OP_i, PR_i), DS_i)$ and srt be a sort such that: $srt \in S_1$, $srt \notin DS_1$ and $srt \in DS_2$ (hence $srt \in S_2$). Then we may specify the static structure of the elements of sort srt in SP_1 and the dynamic one in SP_2; when we consider $\mathrm{SP}_1 + \mathrm{SP}_2$ we obtain the wanted specification.

In practice it is often useful to use a derived construct, **enrich**, defined by:
enrich SP$'$ **by sorts** S **dsorts** DS **opns** OP **preds** PR **axioms** AX $=_{\mathrm{def}}$
let $((S', OP', PR'), DS') = Sig(\mathbf{S}[\mathrm{SP}'])$ in
SP$'$ + **sorts** $S \cup S'$ **dsorts** $DS \cup DS'$ **opns** $OP \cup OP'$ **preds** $PR \cup PR'$ **axioms** AX.

Renaming This construct is used to avoid name-clashes when putting specifications together. We shall consider bijective renamings only, represented by a signature isomorphism (i.e. a bijective signature morphism, see Def. 2).

(•) **rename** SP **with** $\rho \in \mathit{SP\_EXPR}$
for all SP $\in \mathit{SP\_EXPR}$ and all LT-signature isomorphisms ρ;
S[**rename** SP **with** ρ] =
if ρ is an isomorphism from $\mathit{Sig}(\mathbf{S}[\mathrm{SP}])$ into $LT\Sigma'$ then
$(LT\Sigma', \{\mathrm{L}' \mid \mathrm{L}'$ is an $LT\Sigma'$-structure and $\rho^{-1}(\mathrm{L}') \in \mathit{Mod}(\mathbf{S}[\mathrm{SP}])\})$
else undefined,
where if $LT\Sigma = \mathit{Sig}(\mathbf{S}[\mathrm{SP}])$, then $\mathrm{L} = \rho^{-1}(\mathrm{L}')$ is the $LT\Sigma$-structure defined by:
- $\mathrm{L}_{srt} = \mathrm{L}'_{\rho(srt)}$ for all sorts srt of $LT\Sigma$,
- $Op^{\mathrm{L}} = \rho(Op)^{\mathrm{L}'}$ for all operation symbols Op of $LT\Sigma$,
- $Pr^{\mathrm{L}} = \rho(Pr)^{\mathrm{L}'}$ for all predicate symbols Pr of $LT\Sigma$.

Export This construct is used to specify which parts of a specification (sorts, dynamic sorts, operations and predicates) should be "visible from outside". Alternatively, it specifies which parts should be hidden (namely, the non-exported ones).

(•) **export** $LT\Sigma$ **from** SP $\in \mathit{SP\_EXPR}$
for all SP $\in \mathit{SP\_EXPR}$ and all LT-signatures $LT\Sigma$
S[**export** $LT\Sigma$ **from** SP] =
if $LT\Sigma \subseteq \mathit{Sig}(\mathbf{S}[\mathrm{SP}])$, then $(LT\Sigma, \{\mathrm{L}_{|LT\Sigma} \mid \mathrm{L} \in \mathit{Mod}(\mathbf{S}[\mathrm{SP}])\})$
else undefined.

In the examples we use also the dual construct, **hide**, defined by:
hide H **in** SP $=_{\mathrm{def}}$ **export** $LT\Sigma$ **from** SP,
where: H is a set of sorts, dynamic sorts, operation symbols and predicate symbols and $LT\Sigma = \mathit{Sig}(\mathbf{S}[\mathrm{SP}]) - H$.

With these constructs we can act on a dynamic sort, say ds, in two ways: we can hide ds completely (as in the classical setting); or hide its dynamic features only (this can be obtained, when using **export**, by taking $LT\Sigma = (\Sigma, DS)$ where ds appears in Σ but not in DS and Σ does not contain the label sort and the transition predicate for ds).

Reachable This construct is used to restrict the models of a specification to those term-generated (i.e. reachable); here we consider a more general construct which allows us to restrict the models to those "partially term-generated", i.e. where some of the operations act as "constructors".

(•) **reach** SP **on** $C \in \mathit{SP\_EXPR}$
for all SP $\in \mathit{SP\_EXPR}$ and all sets of operation symbols C
S[**reach** SP **on** C] =
if $C \subseteq \mathit{Opns}(\mathit{Sig}(\mathbf{S}[\mathrm{SP}]))$, then
$(\mathit{Sig}(\mathbf{S}[\mathrm{SP}]), \{\mathrm{L} \mid \mathrm{L} \in \mathit{Mod}(\mathbf{S}[\mathrm{SP}])$ and L is C-generated $\})$
else undefined.
(see Appendix A.1 for the definition of "C-generated")

3.3. Conditional LT-specifications

At the design level we consider only conditional LT-specifications, since they admit initial models with interesting properties.

Definition 6. *A* conditional LT-specification *is an LT-specification* $(LT\Sigma, CAX)$, *where* CAX *is a set of conditional formulae (see Appendix A.1).* □

Example 3.1. We give a conditional LT-specification for a simple concurrent calculus, more or less the finite higher-order Milner's CCS.

spec HCCS =
  enrich INT **by**
  sorts *message*
  dsorts *beh*
  opns $Mi\colon int \to message$
    $Mb\colon beh \to message$
    $\tau\colon \to l\text{-}beh$
    $OUT, IN\colon message \to l\text{-}beh$
    $Nil\colon \to beh$
    $\_\,.\,\_\colon l\text{-}beh \times beh \to beh$
    $\_+\_,\ \_\,||\,\_\colon beh \times beh \to beh$
  axioms
  (1) $b_1 + b_2 = b_2 + b_1 \quad (b_1 + b_2) + b_3 = b_1 + (b_2 + b_3)$
  (2) $b + Nil = b \quad b + b = b$
  (3) $b_1 \,||\, b_2 = b_2 \,||\, b_1 \quad (b_1 \,||\, b_2) \,||\, b_3 = b_1 \,||\, (b_2 \,||\, b_3) \quad b \,||\, Nil = b$
  (4) $l\,.\,b \xrightarrow{l} b \quad b_1 \xrightarrow{l} b_1' \supset b_1 + b_2 \xrightarrow{l} b_1' \quad b_1 \xrightarrow{l} b_1' \supset b_1 \,||\, b_2 \xrightarrow{l} b_1' \,||\, b_2$
  (5) $b_1 \xrightarrow{\mathrm{OUT}(m)} b_1' \wedge b_2 \xrightarrow{\mathrm{IN}(m)} b_2' \supset b_1 \,||\, b_2 \xrightarrow{\tau} b_1' \,||\, b_2'$

INT is the usual algebraic specification of integers. The axioms 1, 2 and 3 express static properties of the behaviours; while 4, 5 express dynamic properties of the behaviours by defining their transitions. **End example**

In the following let SP denote a generic conditional LT-specification $(LT\Sigma, CAX)$, where $LT\Sigma = (\Sigma, DS)$ and $\Sigma = (S, OP, PR)$.

Proposition 2. *There exists* I_{SP} *(initial) in* $Mod(\mathrm{SP})$ *characterized by:*

- *for all* $t_1, t_2 \in T_\Sigma$ *of the same sort* $\mathrm{I}_{\mathrm{SP}} \models t_1 = t_2$ *iff* $CAX \vdash t_1 = t_2$;
- *for all* $Pr \in PR$ *and all* $t_1, \ldots, t_n \in T_\Sigma$ *of appropriate sorts* $\mathrm{I}_{\mathrm{SP}} \models Pr(t_1, \ldots, t_n)$ *iff* $CAX \vdash Pr(t_1, \ldots, t_n)$;

where $\vdash$ *denotes first-order provability (see in Appendix A.1 a sound and complete deductive system for conditional formulae).*

Proof. See [41]. □

Example 3.2. Since all axioms of HCCS are conditional formulae, there exists $\mathrm{I}_{\mathrm{HCCS}}$. **End example**

Note that the second part of Prop. 2 holds in particular for the predicates corresponding to the transition relations; thus the transitions of the dynamic elements represented by the initial model of an LT-specification are just the ones logically deducible from the set of axioms CAX.

The above remark suggests another way to characterize I_{SP}. Consider for simplicity the case of specifications with only one dynamic sort, say ds (i.e., $DS = \{ds\}$). Then I_{SP} determines the lts: $((\mathrm{I}_{\mathrm{SP}})_{ds}, (\mathrm{I}_{\mathrm{SP}})_{l\text{-}ds}, \rightarrow)$, with $\rightarrow$ defined by the inductive rules

$$\frac{s_i \xrightarrow{l_i} s_i' \quad i = 1, \ldots, n}{s \xrightarrow{l} s'} \; cond$$

for all $\wedge_{i=1,\ldots,n}\, s_i \xrightarrow{l_i} s_i' \wedge cond \supset s \xrightarrow{l} s' \in CAX$, where $\rightarrow$ does not appear in *cond*.

In the case of conditional LT-specifications the difference between static and dynamic axioms can be made more precise:

- $\wedge_{i=1,\ldots,n}\, \alpha_i \supset \alpha$, where α has form either $t = t'$ or $Pr(t_1, \ldots, t_m)$ with Pr different from the transition predicate is a *static axiom*, while
- $\wedge_{i=1,\ldots,n}\, \alpha_i \supset t \xrightarrow{t'} t''$ is a *dynamic axiom*.

If static and dynamic axioms are separated, then we can have another characterization of the initial model. There are two interesting cases.

Weakly separated The transition predicate does not appear in the static axioms. Assume

$$\mathrm{STAT(SP)} = (LT\Sigma, CAX - \{\text{dynamic axioms in } CAX\})$$

and $\mathrm{SI} = \mathrm{I}_{\mathrm{STAT(SP)}}$ (which exists); then I_{SP} determines the lts: $(\mathrm{SI}_{ds}, \mathrm{SI}_{l\text{-}ds}, \rightarrow)$, with $\rightarrow$ defined by the inductive rules

$$\frac{s_i \xrightarrow{l_i} s_i' \quad i = 1, \ldots, n}{s \xrightarrow{l} s'} \; cond$$

for all dynamic axioms $\wedge_{i=1,\ldots,n}\, s_i \xrightarrow{l_i} s_i' \wedge cond \supset s \xrightarrow{l} s' \in CAX$, where $\rightarrow$ does not appear in *cond*. The difference with the general case is that here the static data may be defined separately from the transition relation; notice that now the metavariables of the inductive rules range over the quotients determined by the static axioms (carriers of SI).

Strongly separated The static axioms are as for weakly separated and the dynamic axioms have form

$$\wedge_{i=1,\ldots,n}\, s_i \xrightarrow{l_i} s_i' \supset s \xrightarrow{l} s'.$$

Here there is no side condition *cond*, and so we can forget the algebraic structure of SI and only consider the two sets SI_{ds} and $\mathrm{SI}_{l\text{-}ds}$.

Clearly the above facts hold also for LT-specifications with several dynamic sorts: they determine a family of labelled transition systems whose transition relations are defined by analogous (multiple) inductive systems.

4. A non-toy example

To show the modularity and the possibility of giving specifications of dynamic elements with very different characteristics of our approach we consider a simple, but non-toy, example of concurrent system with several features, frequently present in systems of interest, as different kinds of cooperation between processes and different kinds of parallelism.

4.1. The concurrent architecture CA

CA is a concurrent architecture with process components interacting both by handshaking communication, broadcasting communication and by writing/ reading a shared buffer.

Below, we informally describe the structure of CA and graphically represent it in Fig. 5, where the ovals represent the active components, the rectangles the passive ones and the arrows the cooperations among the various components.

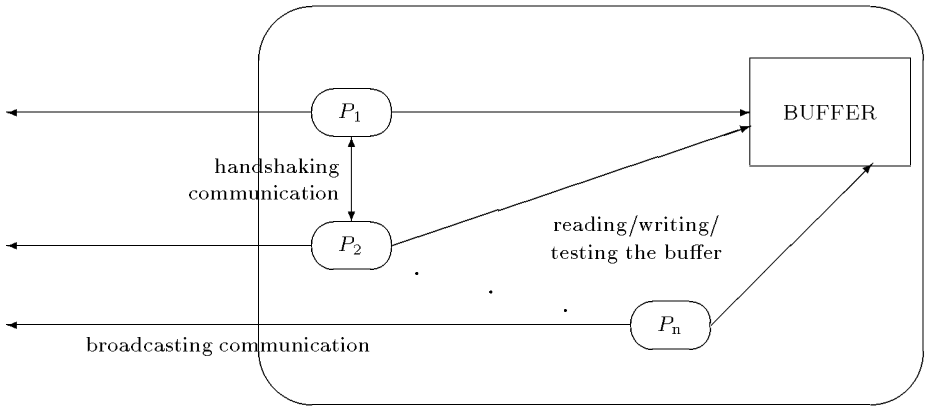
P_1
P_2
P_n
BUFFER
handshaking communication
reading/writing/ testing the buffer
broadcasting communication

Fig. 5. The schematic structure of CA

CA consists of a variable number of processes and a buffer shared among them; "variable" means that processes may terminate and new processes may be created.

The process components have the following features.

Processes communicate among them by exchanging messages in a synchronous mode through channels (handshaking communication) and by reading/ writing messages on the buffer; moreover the processes could also communicate with the world outside CA (consisting of similar architectures) sending and receiving messages in a broadcasting mode; messages are simply values. The values handled by processes may be: integers, booleans, Pascal-like arrays of values, but also the processes themselves are values (that allows e.g. that a process may be received by another one, stored in the local memory and used afterwards for creating a new component of the architecture).

Each process has a (private) local memory and its activity is given by a sequence of commands defined by the following pattern rules.

$$
\begin{array}{lll}
c & ::= & \mathit{Write}(x) \mid \mathit{Read}(x) \mid \mathit{Is\_Empty}(x) \qquad (1)\\
 & & \mathit{Send}(x, ch) \mid \mathit{Rec}(x, ch) \mid \mathit{BSend}(x) \mid \mathit{BRec}(x) \mid \qquad (2)\\
 & & \mathit{Start}(x) \mid \qquad (3)\\
 & & c_1; c_2 \mid c_1 + c_2 \mid \qquad (4)\\
 & & \mathit{Skip} \mid \mathit{seq\text{-}c} \qquad (5)
\end{array}
$$

where x, ch and $seq\text{-}c$ are respectively the nonterminals for variables, channel identifiers and sequential commands.

A CA process may: write the value of a variable on the buffer; read a value from the buffer assigning it to a variable; test if the buffer is empty assigning the result of the test to a variable (1); exchange messages through the channels in a handshaking way and with the world outside CA in a broadcasting way (2); create a new process stored in a variable (recall that processes may be values) (3); perform the sequential composition of two commands and nondeterministically choose between two commands (4); execute sequential commands not further specified (i.e. commands which require neither interactions with other processes, nor with the buffer, nor with the world outside CA) (5).

The buffer is organized as an unbounded queue. We assume that only the testings of the buffer may be performed simultaneously with other kind of accesses (readings and writings).

The processes perform their activity in a completely free parallel mode except for the synchronous interactions required by handshaking communications and the assumptions on simultaneous buffer accesses.

A CA_NET architecture consists of several instances of CA in parallel, which interact by exchanging messages in a broadcasting mode (see Fig. 6). In this case the various components of the net perform their activity in a completely free parallel mode.

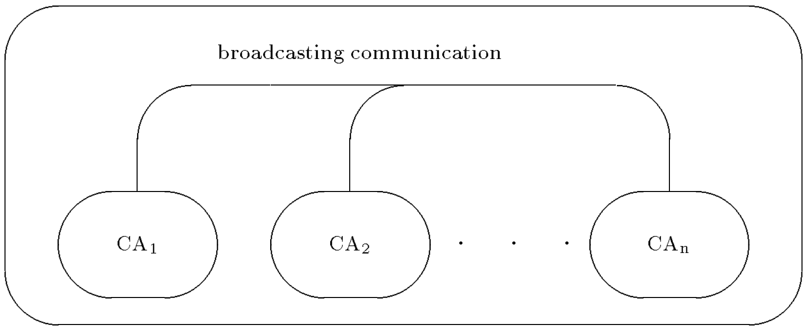

Fig. 6. The schematic structure of CA_NET

4.2. Specification of CA processes

We specify the CA processes introduced in the previous subsection by a design LT-specification PROC (thus with initial semantics and so corresponding to one abstract LT-structure, i.e. to one lts) in a structured way, by giving first the specification of the values handled by the processes, VALUE, and then those of the states and of the labels of the lts, PROC_STATE and PROC_LAB.

The values handled by the CA processes are integers, booleans and Pascal-like arrays of values, but also the processes themselves are values.

spec VALUE =
  enrich INT + BOOL **by**
  sorts $value,\ proc$
  opns $Error{:} \to value$
    $Emi{:}\, int \to value$
    $Emb{:}\, bool \to value$
    $Emp{:}\, proc \to value$
    $\{[\_, \ldots, \_]{:}\, \underbrace{value \times \ldots \times value}_{n \text{ times}} \to value \mid n \geq 2\}$
    $\{Sel_n{:}\, value \to value \mid n \geq 1\}$
  axioms
    $\{Sel_n([v_1, \ldots, v_n, \ldots, v_m]) = v_n \mid 2 \leq m, 1 \leq n \leq m\}$
    $\{Sel_n([v_1, \ldots, v_m]) = Error \mid n > m \geq 2\}$
    $\{Sel_n(Error) = Error, \quad Sel_n(Emi(i)) = Error$
    $Sel_n(Emb(b)) = Error, \quad Sel_n(Emp(p)) = Error \mid 1 \leq n\}$
    $\{[v_1, \ldots, v_{n-1}, Error, v_{n+1}, \ldots, v_m] = Error \mid 2 \leq m, 1 \leq n \leq m\}$

where INT and BOOL are the usual specifications of integers and booleans, *Emi*, *Emb*, *Emp* the operations embedding the basic values into the sort *value* and $[\_, \ldots, \_]$, Sel_n the array values constructors and selectors respectively.

In this specification we do not require anything on the sort *proc*; thus the carrier of such sort in the models of VALUE may be whatever; however this is not a problem, since when VALUE will be used in the following to build up the CA process specification, in any model the sort *proc* will have the right elements (recall that if SP and SP$'$ have both a sort s, then the sum SP + SP$'$ will have one sort s with the properties required by SP and those by SP$'$; moreover if s is static in SP and dynamic in SP$'$ it will be dynamic in the sum, see Sect. 3.2).[1]

The states of the lts modelling the CA processes are pairs consisting of the command to be executed and of the content of the local memory.

spec COMMAND =
  enrich VID + CHID **by**
  sorts $command$
  opns $Write, Read, Is\_Empty, BSend, BRec, Start{:}\, vid \to command$
    $Send, Rec{:}\, vid \times chid \to command$
    $Skip{:} \to command$
    $\_;\_,\ \_+\_{:}\, command \times command \to command$
    $Seq_1{:} \ldots \to command$
    $\ldots$
    $Seq_n{:} \ldots \to command$
  axioms
    $c_1;(c_2;c_3) = (c_1;c_2);c_3 \qquad Skip;c = c \qquad c_1 + c_2 = c_2 + c_1$

The specifications VID (variable identifiers) and CHID (channel identifiers) are not further detailed here, since they are not relevant; Seq_1, ..., Seq_n are the operations defining the sequential commands.

[1] The specification VALUE may be considered a case of "draft specification", i.e. a specification which does not fully determine a significant (w.r.t. the application) abstract data type (in our case it has too many models), but it is useful to split the presentation of the final specification (in this case that of the CA processes) in a nice and sensible way, making it more readable.

spec LMEM = **rename** MAP(VID, *vid*, VALUE, *value*, *Error*) **with** [*lmem*/*map*]

The parametric specification MAP (finite maps) is given in Appendix A.2.

spec PROC_STATE =
  enrich COMMAND + LMEM **by**
  dsorts *proc*
  opns $<\_,\_>\colon command \times lmem \to proc$
    $Seed\colon \to proc$

The labels of the lts modelling the CA processes are defined by the specification PROC_LAB and are in correspondence with the executions of the concurrent commands; except τ which corresponds to the executions of the sequential commands. For simplicity we do not report the specification PROC_LAB; however it is not difficult to deduce its complete definition by looking at the axioms of PROC.

Finally, in the specification PROC we give the activity of the CA processes, by axiomatically defining the transitions of the lts modelling them.

spec PROC =
  enrich PROC_STATE + PROC_LAB **by**
  axioms

(1) $<Write(x), lm> \xrightarrow{\mathrm{WRITE}(lm(x))} <Skip, lm>$

(2) $<Read(x), lm> \xrightarrow{\mathrm{READ}(v)} <Skip, lm[v/x]>$

(3) $<Is\_Empty(x), lm> \xrightarrow{\mathrm{IS\_EMPTY}(v)} <Skip, lm[v/x]>$

(4) $<Send(x, ch), lm> \xrightarrow{\mathrm{SEND}(lm(x), ch)} <Skip, lm>$

(5) $<Rec(x, ch), lm> \xrightarrow{\mathrm{REC}(v, ch)} <Skip, lm[v/x]>$

(6) $<BSend(x), lm> \xrightarrow{\mathrm{B\_SEND}(lm(x))} <Skip, lm>$

(7) $<BRec(x), lm> \xrightarrow{\mathrm{B\_REC}(v)} <Skip, lm[v/x]>$

(8) $lm(x) = Emp(p) \supset <Start(x), lm> \xrightarrow{\mathrm{START}(p)} <Skip, lm>$

(9) $Seed \xrightarrow{\mathrm{CREATE}(p)} p$

(10) $<c_1, lm> \xrightarrow{lp} <c_1', lm'> \supset <c_1; c_2, lm> \xrightarrow{lp} <c_1'; c_2, lm'>$

(11) $<c_1, lm> \xrightarrow{lp} <c_1', lm'> \supset <c_1 + c_2, lm> \xrightarrow{lp} <c_1', lm'>$

(12) $\ldots \supset <Seq_1(\ldots), lm> \xrightarrow{\tau} <\ldots, \ldots>$

$\ldots$

Above "_(_)" and "_[_/_]" denote respectively the selection of the associate element and the map updating operation of the specification MAP (see Appendix A.2).

The axioms 1, 2, 3 define the writing, reading and testing the emptiness of the buffer; 4, 5 the handshaking communications and 6, 7 the broadcasting communications outside CA. The axiom 8 defines the creation of a new process defined by the value contained in the local variable x (if it has type *proc*). The axiom 9 defines the capability of a new process of being created represented by a transition, labelled by *CREATE*(p), of the process *Seed* into the initial state of the newly created process p. The axioms 10, 11 define the sequential composition and the nondeterministic choice; notice that these two axioms are sufficient since ";" is associative, *Skip*; $c = c$ and "+" is commutative (from the static axioms in COMMAND). Several axioms having the form 12, not further detailed here, define the sequential commands.

4.3. Specification of CA

We formally define the lts modelling CA by the LT-specification CA_SPEC given below; also in this case we give it in a structured way, first, active and passive components, then states, labels and finally transitions.

The active components The active components of CA are the process and are described by the LT-specification PROC given in subsection 4.2.

The passive component The passive component of CA is the buffer, in this case a queue; its states are given by the following (static) specification of an abstract data type.

spec BUFFER =
  enrich VALUE **by**
  sorts $buffer$
  opns $Empty\colon \rightarrow buffer$
    $Put\colon value \times buffer \rightarrow buffer$
    $First\colon buffer \rightarrow value$
    $Get\colon buffer \rightarrow buffer$
  preds $Is\_Not\_Empty\colon buffer$
  axioms
  (1) $First(Empty) = Error$ $First(Put(v, Empty)) = v$
  (2) $Is\_Not\_Empty(bf) \supset First(Put(v, bf)) = First(bf)$
  (3) $Get(Empty) = Empty$ $Get(Put(v, Empty)) = Empty$
  (4) $Is\_Not\_Empty(bf) \supset Get(Put(v, bf)) = Put(v, Get(bf))$
  (5) $Is\_Not\_Empty(Put(v, bf))$

The operation *Empty* denotes the empty queue, *Put* is the operation for adding a value on the queue and *First*, *Get* those for getting and dropping the first element from a queue.

The labels The labels of the lts modelling CA keep trace of the values either sent to or received from the external world by broadcasting communications during every transition; thus we have

spec CA_LAB =
  enrich VALUE **by**
  sorts $l\text{-}ca$
  opns $\tau\colon \rightarrow l\text{-}ca$
    $BS, BR\colon value \rightarrow l\text{-}ca$
    $\_ \,\&\, \_\colon l\text{-}ca \times l\text{-}ca \rightarrow l\text{-}ca$
  axioms
    $l_1 \,\&\, l_2 = l_2 \,\&\, l_1$ $(l_1 \,\&\, l_2) \,\&\, l_3 = l_1 \,\&\, (l_2 \,\&\, l_3)$

The activity The transitions of the lts modelling CA (i.e. the CA activity) are modularly defined by first giving some "partial moves" of groups of processes, corresponding to sets of process synchronous cooperations (i.e. sets of process transitions which must be performed together) that in turn may be performed together; then these partial moves are used to give the CA transitions. The partial moves are still formalized as labelled transitions (defined by an auxiliary predicate $\_ \overset{-}{\rightsquigarrow} \_$) whose labels contain information on the buffer accesses performed in the move and on the communications with the external world (defined by the specification AUX given below). *No_B_Access* is a predicate checking whether a partial move does not involve a buffer access (either reading or writing it).

spec AUX =
  enrich VALUE **by**
  sorts aux
  opns $\tau \colon \to aux$
  $WRITE, READ, IS\_EMPTY, B\_SEND, B\_REC \colon value \to aux$
  $\_ \mathbin{//} \_ \colon aux \times aux \to aux$
  preds $No\_B\_Access \colon lab$
  axioms
  $l_1 \mathbin{//} l_2 = l_2 \mathbin{//} l_1 \qquad (l_1 \mathbin{//} l_2) \mathbin{//} l_3 = l_1 \mathbin{//} (l_2 \mathbin{//} l_3)$
  $No\_B\_Access(\tau) \qquad No\_B\_Access(IS\_EMPTY(v))$
  $No\_B\_Access(B\_SEND(v)) \qquad No\_B\_Access(B\_REC(v))$
  $No\_B\_Access(l_1) \wedge No\_B\_Access(l_2) \supset No\_B\_Access(l_1 \mathbin{//} l_2)$

Below MSET is the parametric specification of finite multiset reported in Appendix A.2.

spec CA_SPEC =
  enrich rename MSET(PROC, *proc*) **with** [*proc_mset*/*mset*] + BUFFER + CA_LAB + AUX **by**
  dsorts ca
  opns $\_ \mid \_ \colon proc\_mset \times buffer \to ca$
  $Out \colon aux \to l\text{-}ca$
  preds $\_ \overset{-}{\rightsquigarrow} \_ \colon ca \times aux \times ca$
  axioms

(1) $p \xrightarrow{\tau} p' \supset p \mid bf \overset{\tau}{\rightsquigarrow} p' \mid bf$

(2) $p \xrightarrow{\mathrm{READ}(v)} p' \wedge First(bf) = v \supset p \mid bf \overset{\mathrm{READ}(v)}{\rightsquigarrow} p' \mid Get(bf)$

(3) $p \xrightarrow{\mathrm{IS\_EMPTY}(False)} p' \wedge Is\_Not\_Empty(bf) \supset p \mid bf \overset{\mathrm{IS\_EMPTY}(False)}{\rightsquigarrow} p' \mid bf$

(4) $p \xrightarrow{\mathrm{IS\_EMPTY}(True)} p' \supset p \mid Empty \overset{\mathrm{IS\_EMPTY}(True)}{\rightsquigarrow} p' \mid Empty$

(5) $p \xrightarrow{\mathrm{WRITE}(v)} p' \supset p \mid bf \overset{\mathrm{WRITE}(v)}{\rightsquigarrow} p' \mid Put(v, bf)$

(6) $p_1 \xrightarrow{\mathrm{REC}(v,ch)} p_1' \wedge p_2 \xrightarrow{\mathrm{SEND}(v,ch)} p_2' \supset p_1 \mid p_2 \mid bf \overset{\tau}{\rightsquigarrow} p_1' \mid p_2' \mid bf$

(7) $p_1 \xrightarrow{\mathrm{START}(p)} p_1' \wedge p_2 \xrightarrow{\mathrm{CREATE}(p)} p_2' \supset p_1 \mid p_2 \mid bf \overset{\tau}{\rightsquigarrow} p_1' \mid p_2' \mid bf$

(8) $p \xrightarrow{\mathrm{B\_SEND}(v)} p' \supset p \mid bf \overset{\mathrm{B\_SEND}(v)}{\rightsquigarrow} p' \mid bf$

(9) $p \xrightarrow{\mathrm{B\_REC}(v)} p' \supset p \mid bf \overset{\mathrm{B\_REC}(v)}{\rightsquigarrow} p' \mid bf$

(10) $pms_1 \mid bf \overset{l_1}{\rightsquigarrow} pms_1' \mid bf \wedge No\_B\_Access(l_1) \wedge pms_2 \mid bf \overset{l_2}{\rightsquigarrow} pms_2' \mid bf' \supset$
  $pms_1 \mid pms_2 \mid bf \overset{l_1 \mathbin{//} l_2}{\rightsquigarrow} pms_1' \mid pms_2' \mid bf'$

(11) $pms \mid bf \overset{l}{\rightsquigarrow} pms' \mid bf' \supset pms \mid pms_1 \mid bf \xrightarrow{\mathrm{Out}(l)} pms' \mid pms_1 \mid bf'$

(12) $Out(\tau) = \tau \qquad Out(WRITE(v)) = \tau \qquad Out(READ(v)) = \tau$
  $Out(IS\_EMPTY(v)) = \tau$

$Out(B\_SEND(v)) = BS(v) \qquad Out(B\_REC(v)) = BR(v)$
$Out(l_1 \,//\, l_2) = Out(l_1)\ \&\ Out(l_2)$

The axiom 1 states that each process internal action capability becomes a partial move; the axiom 2 that an action capability of form $READ(v)$ becomes a partial move only if the first available element of the buffer is v; the axiom 5 that an action capability of form $WRITE(v)$ always becomes a partial move putting the value v on the buffer and the axioms 3, 4 that an action capability of form $IS\_EMPTY(True)$ [$IS\_EMPTY(False)$] becomes a partial move iff the buffer is empty [not empty].

Handshaking communication is a partial move involving two processes: the one which sends the message and the one which receives it (axiom 6).

The axiom 7 defines the creation of a new process; note that in each application of this axiom p_2 is equal to *Seed* and p'_2 to p; then we have $p_1 \mid pms \mid bf$ $= p_1 \mid Seed \mid pms \mid bf \xrightarrow{\tau} p_1 \mid p \mid pms \mid bf$, where pms is a multiset of process states. The axioms 8 and 9 state that the action capabilities corresponding to broadcasting communications, can always become partial moves. The axiom 10 states that each partial move "$pms_1 \mid bf \overset{l_1}{\leadsto} pms'_1 \mid bf$" not involving a buffer access (either reading or writing it) may be performed together with whatever other group of partial moves; as a consequence we have that a buffer access cannot be performed together with another buffer access. The axiom 11 says that each group of partial moves may become a transition of CA; notice that here pms_1 is the set of the process components which do not take part in the transition (recall that CA is a free parallel system); the associated interaction with the external world is given by the broadcasting communications present in such transition (formalized by the operation *Out* defined by the axioms 12).

4.4. Specification of variants of CA

In this subsection, to show the modular features of LTL/SMoLCS we give some hints on how modifications of the assumptions on CA reported in Sect. 4.1 may be reflected in the specification CA_SPEC by changing some of its parts; ([15] presents the specifications of a full set of CA variations).

Values If the values handled by the processes are modified, then it is sufficient to replace the specification VALUE with another one having the static sort *value* (e.g. array-like values may be replaced with LISP-like lists).

For example, to drop the higher-order features of CA it is sufficient to drop the sort *proc* and the operation *Emp* from the original specification VALUE.

Buffer If the buffer component of the architecture is modified, then it is sufficient to replace the specification BUFFER with another one whose signature includes that of the original BUFFER.

For example, if the buffer becomes an unbounded stack, then we replace the axioms 1, ..., 4 of BUFFER with:

$First(Empty) = Error \qquad First(Put(v, bf)) = v$
$Get(Put(v, bf)) = bf \qquad Get(Empty) = Empty;$

while if it becomes a cell which can contain at most one value, then the new axioms are

$Put(v, Put(v', bf)) = Put(v, bf)$
$First(Put(v, Empty)) = v \quad First(Empty) = Error$
$Get(Put(v, Empty)) = Empty \quad Get(Empty) = Empty$

Buffer access If the assumptions on the allowed simultaneous buffer accesses are modified, then it is sufficient to change the axioms defining the auxiliary predicate *No_B_Access* of AUX.

For example, if also the testing of the buffer is considered an access, then the new axioms are

$No\_B\_Access(\tau)$
$No\_B\_Access(B\_SEND(v))$
$No\_B\_Access(B\_REC(v))$
$No\_B\_Access(l_1) \land No\_B\_Access(l_2) \supset No\_B\_Access(l_1 \,//\, l_2)$

Parallel activity Instead, if the assumptions on how the processes perform their activities in parallel are modified, then it is sufficient to change the axiom 11.

For example, if the CA becomes an interleaving system, then the axioms replacing 11 are:

$$pms \mid bf \overset{lt}{\leadsto} pms' \mid bf' \supset pms \mid pms_1 \mid bf \xrightarrow{Out(lt)} pms' \mid pms_1 \mid bf'$$

for all terms $lt \in \{ \tau, B\_SEND(v), B\_REC(v), WRITE(v), READ(v), IS\_EMPTY(v) \}$.

4.5. LT-specification of CA_NET

We formally define CA_NET with the LT-specification CA_NET_SPEC given below.

The active components of CA_NET are a multiset of (a variant of) CA architectures described by the LT-specification CA_SPEC given either in subsection 4.3 or 4.4; while CA_NET has no passive components. Since CA_NET is a closed system (no interactions with the external world) its transitions are all labelled by τ. In CA_NET only one broadcasting communication may be performed at each time, while there are no restrictions on the internal activities of the component architectures, so its activity is simply defined below.

spec CA_NET_SPEC =
  enrich rename MSET(CA_SPEC, *ca*) **with** [*ca_net*/*mset*] **by**
  dsorts *ca_net*
  opns $\tau \colon \to lab\text{-}ca\_net$
  preds $\_ \xrightarrow{\_} \_ \colon ca\_net \times lab\text{-}ca\_net \times ca\_net$
  axioms

(1) $\{c \xrightarrow{B\_SEND(v)} c' \land c_1 \xrightarrow{B\_REC(v)} c'_1 \land \ldots \land c_n \xrightarrow{B\_REC(v)} c'_n \land c_{n+1} \xrightarrow{\tau} c'_{n+1} \land \ldots \land c_m \xrightarrow{\tau} c'_m \supset c \mid c_1 \mid \ldots \mid c_n \mid c_{n+1} \mid \ldots \mid c_m \mid cms \xrightarrow{\tau} c' \mid c'_1 \mid \ldots \mid c'_n \mid c'_{n+1} \mid \ldots \mid c'_m \mid cms \mid n, m \geq 0\}$

(2) $\{c_1 \xrightarrow{\tau} c'_1 \land \ldots \land c_n \xrightarrow{\tau} c'_n \supset c_1 \mid \ldots \mid c_n \mid cms \xrightarrow{\tau} c'_1 \mid \ldots \mid c'_n \mid cms \mid n \geq 1\}$

The axiom 1 states that each architecture may send a value by a broadcasting communication, which may be received by whatever number of other architectures (also none), while other architectures (also none) perform internal activities. The axiom 2 states that whatever number of architectures (at least one) may perform internal activities. In both cases, *cms* is the multiset (possibly empty) of the architectures which stay idle.

Notice that a CA architecture may also perform several broadcasting communications simultaneously; but these capabilities will never become effective in this kind of CA_NET.

5. LT-specifications: requirement level

5.1. First-order LT-specifications

An LT-specification (see Def. 5) with loose semantics is a way to specify (abstract) requirements about dynamic elements formally modelled by LT-structures.

Indeed let SP = $(LT\Sigma, AX)$, where AX is a set of first-order formulae on $LT\Sigma$, be an LT-specification, then the class of the abstract (i.e. isomorphism classes of) $LT\Sigma$-structures contained in *Mod*(SP) formally defines all dynamic elements satisfying the requirements expressed by SP.

The common and relevant requirements can be distinguished as follows.

Static requirements, i.e. about the structure of the dynamic elements, of their active/passive components and of the handled data; they are expressed by the form of the LT-signature $LT\Sigma$ (e.g. how many types of dynamic elements there are and how they are put together to build a concurrent dynamic element) and by the static axioms (e.g. the parallel combinator is commutative, the order of the active components does not matter).

Dynamic requirements, i.e. about the activity of the dynamic elements; they are expressed by the static axioms about the label data, corresponding to requirements on their interactions with their external world, see Sect. 2.1 for the role of labels (e.g. no interaction with the external world is formalized by saying that there is just one constant label) and by the dynamic axioms (about labelled transitions), i.e. in which the transition predicates appear (e.g., a dynamic element in a certain state has just two capabilities of moving to two other states or it cannot have some capabilities).

Example 5.1. Requirements on concurrent architectures
Here we show how to formalize abstract requirements on classes of concurrent architectures similar to CA, see Sect. 4, by means of first-order LT-specifications with loose semantics. Also in this case we proceed in a modular way, by giving first the requirements on the component processes and after on how they cooperate.

spec PROC_REQ =
    dsorts *proc*
    sorts *value*
    opns $Value\colon proc \rightarrow value$
        $WRITE, READ, OUT, IN\colon value \rightarrow l\text{-}proc$
        $\tau\colon \rightarrow l\text{-}proc$

axioms
(1) $\mathit{Value}(p) = \mathit{Value}(p') \supset p = p'$
(2) $\mathit{WRITE}(v_1) \neq \mathit{READ}(v_2) \neq \mathit{OUT}(v_3) \neq \mathit{IN}(v_4) \neq \tau$
(3) $\mathit{WRITE}(v) = \mathit{WRITE}(v') \supset v = v'$
... analogous axioms for $\mathit{READ}, \mathit{OUT}, \mathit{IN}$ ...
(4) $p \xrightarrow{\mathrm{OUT}(\mathrm{Value}(p_1))} p' \vee p \xrightarrow{\mathrm{IN}(\mathrm{Value}(p_1))} p' \supset \exists\, lp, p_1' \,.\, p_1 \xrightarrow{lp} p_1'$

The above specification requires that the processes can at least perform five different kinds of actions (described by the various operations of sort *l-proc*) using values which may be the same processes (by the operation *Value*). The axiom 4 states that only non-stopped processes can be communicated.

spec CA_REQ =
reach
enrich rename MSET(PROC_REQ, *proc*) **with** [*proc_mset*/*mset*] **by**
dsorts *ca*
sorts *buffer*
opns $\mathit{Empty}, \mathit{Broken}\colon \to \mathit{buffer}$
$\_ \mid \_\colon \mathit{proc\_mset} \times \mathit{buffer} \to \mathit{ca}$
axioms
(1) $(\nexists\, lc, c \,.\, pms \mid bf \xrightarrow{lc} c) \supset (\nexists\, pms', p, p', lp \,.\, pms = p \mid pms' \wedge p \xrightarrow{lp} p')$
(2) $(\nexists\, c, lc \,.\, c \xrightarrow{lc} pms \mid bf) \supset bf = \mathit{Empty}$
(3) $\nexists\, c, lc \,.\, pms \mid \mathit{Broken} \xrightarrow{lc} c$
on
{ $\_ \mid \_\colon \mathit{proc\_mset} \times \mathit{buffer} \to \mathit{ca}$, $\emptyset\colon \to \mathit{proc\_mset}$,
$\{\_\}\colon \mathit{proc} \to \mathit{proc\_mset}$, $\_ \mid \_\colon \mathit{proc\_mset} \times \mathit{proc\_mset} \to \mathit{proc\_mset}$ }

The reachability combinator formalizes a requirement on the structure of the architectures; precisely, "there are several active components of the same sort *proc* and a passive component of sort *buffer*". The axioms of the parametric specification MSET (see Appendix A.2) formalize that the order of the process components is not relevant. No deadlocks are allowed in the specified architectures (axiom 1). Initially the buffer must be empty (axiom 2). The broken buffer stops the whole system (axiom 3).

From the axioms of CA_REQ we can prove (using usual first-order stuff) that the axioms 1 and 3 imply $\forall\, p \,.\, \nexists\, p', lp \,.\, p \xrightarrow{lp} p'$, i.e., that all processes are terminated, which makes the above specification not very interesting. So we can replace axiom 1 with

$$(1') \qquad (bf \neq \mathit{Broken} \wedge \nexists\, lc, c \,.\, pms \mid bf \xrightarrow{lc} c) \supset$$
$$(\nexists\, pms', p, p', lp \,.\, pms = p \mid pms' \wedge p \xrightarrow{lp} p')$$

no deadlocks are allowed in normal situations (when the buffer is ok). **End example**

5.2. More expressive logic

First-order logic allows to express only few requirements on dynamic elements, similar to those given in Ex. 5.1, as properties of the static parts (e.g. on the labels of the processes), limited properties on the concurrent structure (e.g. the order of the process components of an architecture is not relevant) and limited properties on the activity of the dynamic elements (e.g. a terminated process cannot be communicated). A great number of more interesting and relevant properties, as the following one, cannot be expressed.

1. only processes that in any case will terminate may be communicated;
2. during each possible computation each architecture eventually will reach a situation where the buffer is empty;
3. the broken buffer cannot be repaired (i.e. if the buffer becomes *Broken*, then it cannot change its state anymore);
4. no deadlocks are allowed but without fixing the structure of the architectures (i.e. we want to give a specification with admits, as models, for example, both architectures with and without passive components).

First-order LTL may be extended in various ways to cope with properties like the above ones (including classical safety/liveness properties and abstract properties on the concurrent structure).

For what concerns dynamic properties, [34, 35] present an integration of the combinators of branching-time temporal logic in LTL for expressing the safety/ liveness properties; while [59, 18] propose a new logic based on the concept of "abstract event" for expressing overall properties on the activity of the dynamic elements (e.g. a certain activity, not necessarily atomic [i.e. consisting of one state/transition], will be eventually followed by the sequential composition of two other non-atomic activities), and [33] shows an integration with the permission/ obligation combinators of the deontic logic.

Instead, for the "structural properties" entity specifications have been proposed (see [58, 16]) where it is possible to express properties on the concurrent/ distributed structure by using a special predicate "*Are_Sub*" s.t. "*dset Are_Sub d*" is true whenever *dset* (a set of dynamic elements) are all the active (dynamic) components of *d* (a dynamic element).

Using the LTL variants proposed in the papers quoted above, the properties 1, ..., 4 may be formalized as follows, where ⧫, ■ are the usual temporal combinators "eventually" and "always" including the present and $\triangle$ is the CTL-style temporal combinator "for all paths". Notice that, since here we are handling types of dynamic elements, the formulae built by $\triangle$ are anchored to a term of dynamic sort (representing the element to whom the temporal property is referring).

1 $(p \xrightarrow{OUT(Value(p_1))} p' \vee p \xrightarrow{IN(Value(p_1))} p') \supset \triangle(p_1, \blacklozenge[\lambda x.\ \nexists\, p'', lp.x \xrightarrow{lp} p''])$
(a process p_1 is terminating iff in each case it will reach a final situation).

2 $\triangle(c, \blacklozenge[\lambda x\ .\ \exists\ pms\ .\ x = pms \mid Empty])$

3 $\triangle(mp \mid Broken, \blacksquare\,[\lambda x\ .\ \exists\ pms'\ .\ x = pms' \mid Broken])$

4 $(\nexists\, c', lc\ .\ c \xrightarrow{lc} c') \wedge dset\ Are\_Sub\ c \wedge d \in dset \supset \nexists\, d', l'\ .\ d \xrightarrow{l} d'$

6. Implementation of LT-specifications

We extend to the case of LT-specifications the well-known general notion of implementation for algebraic specification of abstract data types due to Sannella and Wirsing (see [64]); we have chosen this notion since it has been proved well adequate in the case of usual static specifications.

Let SP and SP′ be two classical logical/algebraic specifications; when is SP implemented by SP′? There are, at least, two criteria to consider:

- implementing means refining, thus SP′ must be a refinement of SP, i.e. "things" not fixed in SP are made precise in SP′ by adding further requirements;
- implementing means realizing the data and the operations abstractly specified in SP, by using the data and the operations of SP′.

Formally, we have that SP is *implemented by* SP′ *with respect to* f, a function from specifications into specifications, iff $Mod(f(\mathrm{SP}')) \subseteq Mod(\mathrm{SP})$.

The function f describes how the parts of SP are realized in SP′ (implementation as realization); while implementation as refinement is obtained by requiring inclusion of the classes of models.

Clearly not all specification functions are acceptable; for example if f is the constant function returning SP we have a kind of trivial implementation. However, the definition above includes as particular cases the various definitions proposed in the literature. Usually f is a combination of the various operations for structuring specifications proposed in Sect. 3.2. When f is a composition of a renaming, an enrichment with derived operations and predicates, an export and an enrichment with axioms, we have the so called implementation by rename-enrich-restrict-identify of [37, 38]; which corresponds, within the framework of abstract data types, to Hoare's idea of implementation of concrete data types.

The above definition, when used in our setting, yields a reasonable notion of implementation for LT-specifications.

Definition 7. *Let* SP $= (LT\Sigma, AX)$ *and* $\mathrm{SP}_1 = (LT\Sigma_1, AX_1)$ *be two LT-specifications (either design or requirement) and* f *a function from specifications with signature* $LT\Sigma$ *into specifications with signature* $LT\Sigma_1$ *defined by composing specification operations as those of Sect. 3.2.*

SP is implemented by SP_1 via f *(written* SP $\rightsquigarrow\!\!> ^{f}$ SP_1*) iff*
$Mod(f(\mathrm{SP}_1)) \subseteq Mod(\mathrm{SP})$. □

If we impose some conditions on the function f we get particular types of implementations; for example:

- f does not add axioms defining the transition predicates of SP′; then we have a "static implementation", which concerns just the static parts of the specification (for example, either the data handled by the dynamic elements or the states and the labels of the lts modelling them);
- f redefines the transitions of SP′ by composing them sequentially, i.e. by adding axioms like

$$s_1 \xrightarrow{l_1} s_2 \;\wedge\; s_2 \xrightarrow{l_2} s_3 \;\wedge\; \ldots \;\wedge\; s_{n_1} \xrightarrow{l_{n-1}} s_n \;\supset\; s_1 \xRightarrow{l} s_n$$

  ($\longrightarrow$ transition predicate in SP′ and $\Longrightarrow$ transition predicate in SP); we have an "action refining implementation", because the transitions of the dynamic elements of SP are realized by sequences of transitions in SP′.
- f does not change dynamic sorts into static ones: we have a "dynamism-preserving" implementation.

Once we have defined a notion of implementation, it is interesting to study its relations with the specification structure. We limit ourselves to implementations where f is the identity function, using for this the notation SP $\rightsquigarrow\!\!>$ SP′

(notice that in this case SP and SP$'$ must have the same signature). This is not a restriction: from the properties of $\rightsquigarrow$ we can derive properties about $\rightsquigarrow^{f}$, because f is defined as a combination of the structuring operations; indeed, SP $\rightsquigarrow^{f}$ SP$'$ iff SP $\rightsquigarrow f(\mathrm{SP}')$. It is easy to verify the following

Fact 6.1.

1. $\rightsquigarrow$ is a partial order (i.e. it is reflexive, antisymmetric and transitive).
2. All specification operators are monotonic w.r.t. $\rightsquigarrow$: if SP, SP$'$, SP$''$ are specifications, $LT\Sigma \subseteq \mathit{Sig}(\mathrm{SP})$, ρ is a signature isomorphism and SP $\rightsquigarrow$ SP$'$, then
   SP + SP$''$ $\rightsquigarrow$ SP$'$ + SP$''$;
   export $LT\Sigma$ **from** SP $\rightsquigarrow$ **export** $LT\Sigma$ **from** SP$'$;
   rename SP **with** ρ $\rightsquigarrow$ **rename** SP$'$ **with** ρ;
   reach SP **on** C $\rightsquigarrow$ **reach** SP$'$ **on** C. □

Example 6.1. Implementations of CA_REQ$'$
We consider some implementations of the requirement LT-specification CA_REQ$'$ (with the axiom 1$'$) introduced in Ex. 5.1, since there is no room to introduce completely new and more interesting examples.

First, notice that CA_SPEC is not one of its correct (reasonable) implementations. Indeed, let define a function over specifications f s.t. f(CA_SPEC) is given:

- by saying how the various things present in CA_REQ are realized using those of CA_SPEC:
  - by enriching CA_SPEC with the operations
    $\mathit{Broken}\colon \to \mathit{buffer}$, $\quad OUT, IN\colon \mathit{value} \to \mathit{l\text{-}proc}$
    defined by the axioms
    $\mathit{Broken} = \mathit{bf}$, for all bf including an Error value,
    $SEND(v) = OUT(v)$, $B\_SEND(v) = OUT(v)$,
    $REC(v) = IN(v)$, $B\_REC(v) = IN(v)$
  - by renaming *Value* as *Emp*;
- by hiding all operations and sorts present in CA_SPEC and not in CA_REQ$'$ (so the signatures of f(CA_SPEC) and CA_REQ are equal).

Then, we have to check if the axioms of CA_REQ hold in f(CA_SPEC). Unfortunately, the axiom 1$'$ (requiring the absence of deadlocks when the buffer is not broken) does not hold, since all states of CA having only one process component with command part $\mathit{Send}(x, ch)$ are deadlock situations.

Not even PROC is a correct implementation of PROC_REQ, since in PROC whatever process may be communicated outside by either handshaking or broadcasting communications.

Now, we give a design LT-specification describing a particular concurrent system similar to Milner's SCCS having all properties required by the specification CA_REQ$'$. In such specification the elements of various sorts (static and dynamic) as *value*, *proc* and *ca* are completely defined; moreover also the activity of the dynamic elements is fully defined. As usual, first we give the specification of active components, BEH, and after of the whole system, SYSTEM.

spec BEH =
  sorts *chan*, *value*

dsorts beh
opns $Alpha, Beta, \ldots : \rightarrow chan$
$Emb : beh \rightarrow value$
$Emc : chan \rightarrow value$
$OUT, IN : value \rightarrow l\text{-}beh$
$\tau : \rightarrow l\text{-}beh$
$Nil : \rightarrow beh$
$\_?\_, \_!\_ : chan \times beh \rightarrow beh$
$\_ + \_ : beh \times beh \rightarrow beh$ (comm., assoc., id: Nil)
$\delta\ \_ : beh \rightarrow beh$
axioms
$ch!b \xrightarrow{\mathrm{OUT}(Emc(ch))} b \qquad ch?b \xrightarrow{\mathrm{IN}(Emc(ch))} b$
$b_1 \xrightarrow{l} b_1' \supset b_1 + b_2 \xrightarrow{l} b_1'$
$\delta\, b \xrightarrow{\tau} \delta\, b \qquad b \xrightarrow{l} b' \supset \delta\, b \xrightarrow{l} b'$

BEH is an implementation of PROC_REQ; indeed it is sufficient to rename *beh* into *proc* and *Emb*: *beh* $\rightarrow$ *value* into *Value*; to add the operations *READ*, *WRITE*: *value* $\rightarrow$ *l-proc* and to hide all operations which are not needed in PROC_REQ. Then all properties defined by the PROC_REQ axioms are satisfied, since BEH behaviours never communicate other behaviours outside (no transition labelled by either $OUT(b)$ or $IN(b)$ will ever performed).

spec SYSTEM =
enrich rename MSET(BEH, *beh*) **with** [*proc_mset*/*mset*] **by**
sorts $status$
dsorts $system$
opns $Working, Broken : \rightarrow status$
$\_ \mid \_ : proc\_mset \times status \rightarrow system$
$TAU : \rightarrow l\text{-}system$
$IN, OUT : value \rightarrow l\text{-}system$
axioms
$b \xrightarrow{\tau} b' \supset b \mid Working \xrightarrow{\mathrm{TAU}} b' \mid Working$
$b_1 \xrightarrow{\mathrm{OUT}(ch)} b_1' \wedge b_2 \xrightarrow{\mathrm{IN}(ch)} b_2' \supset b_1 \mid b_2 \mid Working \xrightarrow{\mathrm{TAU}} b_1' \mid b_2' \mid Working$
$b_1 \xrightarrow{\mathrm{OUT}(ch)} b_1' \supset b_1 \mid Working \xrightarrow{\mathrm{OUT}(ch)} b_1' \mid Working$
$b_1 \xrightarrow{\mathrm{IN}(ch)} b_1' \supset b_1 \mid Working \xrightarrow{\mathrm{IN}(ch)} b_1' \mid Working$
$mb_1 \mid Working \xrightarrow{ls} mb_1' \mid Working \supset mb_1 \mid mb \mid Working \xrightarrow{ls} mb_1' \mid mb \mid Working$
$mb \mid Working \xrightarrow{\mathrm{TAU}} mb \mid Broken$

SYSTEM is an implementation of CA_REQ; indeed after making the appropriate renamings (including *Working* into *Empty*), enrichments and hidings we get a new specification where all axioms of CA_REQ are satisfied, thus whose models are included into the models of CA_REQ; the axiom $1'$ holds since no behaviour activity may be blocked when the status is *Working*; the axiom 2 holds since there are no initial states and the axiom 3 holds since all axioms of SYSTEM defining transitions require the status to be equal to *Working*.
End example

The above example is a case where the implementing dynamic elements have exactly the same concurrent structure of the implemented ones. However, the notion of implementation of Def. 7 does not imply that restriction, as the requirement LT-specification CA_REQ2, given below, shows, where the elements of sort *proc* are implemented by groups of other processes interacting between them.

spec CA_REQ2 =
  enrich CA_REQ **by**
  dsorts *agent*
  opns $\_ : agent \to proc$
  $\_ \mid\mid\mid \_ : agent \times proc \to proc$
  axioms
  ...

In this case CA_REQ $\rightsquigarrow^{f}$ CA_REQ2, where f is just the hiding of the dynamic sort *agent* and of the operation $\mid\mid\mid$.

7. Relationship with other approaches

7.1. Generalities

In the following we briefly discuss the relationship of LTL/SMoLCS with other methods for the specification of concurrent systems.

The nice idea of modelling processes with labelled transition systems, adopted by LTL/SMoLCS, has been especially advocated by G. Plotkin and R. Milner in their landmark papers [55, 56, 49] on SOS and CCS. Indeed, our method can be seen as a generalization of some very nice features of CCS and SOS, but there are many major differences.

SOS provides indeed a method of defining systems; however it is not specifically targeted at concurrent systems and thus does not provide any support for structural concurrent specifications. Hence an SOS specification of a concurrent system may be not driven by its concurrent structure. For example, in [56] a binary parallel operator gives the structure of a CSP program, while the true concurrent structure is that of an unordered group of processes (i.e., a set); and [46] presents an SOS semantics for a subset of Ada, where the handshaking communication between two tasks (concurrent activity) and the raising of an exception within a task (sequential activity) are modelled in the same way, since there is no way to explicitly distinguish the concurrent and the sequential combinators. Finally, clearly SOS does not provide any means for abstract algebraic specifications; in particular it is not possible to define static axioms, which have been proved so useful and are becoming more and more popular (see [6] for a first appearance and [48] for further strong motivations).

Note also that LTL/SMoLCS is very different from all specification methods based on some particular language, say CL, with a fixed set of primitives for concurrency, where the specification of a system S is a program of CL "corresponding" to S. Those methods work well whenever the concurrent features of S are similar to those of CL; otherwise we have a kind of translation. Consider, for example, specifying a system, say SHM, where the processes interact only by accessing a shared memory and using CCS as specification language (where processes interact only by handshaking communication along channels); then the specification of SHM is a CCS program where some process simulates the shared memory, and so in the end we have more a kind of implementation, than an (abstract) specification. Using our approach, instead, the concurrent features of a system are defined directly, and not implemented by other constructs.

In the literature there are other methods using logical/algebraic techniques for the specification of concurrency (for a more detailed review see [17]); note that here we consider "methods" and not particular instances of algebraic specification of concurrent systems (e.g., an algebraic definition of an ACP calculus, see [24]).

Most of the known methods aim at providing algebraic specifications of the static structures used by processes; among them LOTOS [44] and various versions of "algebraic" Petri nets (see e.g. [62]); all use a fixed concurrent language (CCS, CSP) or a fixed concurrent structure (Petri nets) integrated with abstract data type specifications.

A nice method which gives some support also for specifying different communication schemas is PSF [47], built around some variation (among the many) of ACP; in particular, differently from original ACP, it adopts a transition semantics approach as in CCS. It provides a rich toolset, it is limited to design specifications and does not exploit the idea of process as data, for example to deal with higher-order concurrency.

The apparently closest approach to ours is the "Rewriting Logic" (shortly RL) of Meseguer [48], see [21] for an extensive study of the relationship between the two formalisms. RL specifications roughly correspond to the subset of the conditional LT-specifications where the transitions are not labelled, the static and the dynamic axioms are completely separated and the axioms for closing transitions by congruence, transitivity and reflexivity (here the name "rewriting") are assumed. To be precise, the RL models do not correspond just to (first-order versions) of plain transition systems, but to systems whose transitions are decorated by "proofs", i.e. descriptions of how the transitions has been deduced by the specification axioms; and such proofs can be interpreted as descriptions of how the system components have determined such moves (they can be used e.g. to speak about fairness). But this interesting feature of RL has never been exploited in its applications, and whenever such additional information on transitions are needed, we can encompass them also in LTL/SMoLCS by defining the data type of the proofs and using a 4-ary transition predicate instead of a ternary one. Moreover in RL there is no provision for observational semantics and it is difficult to see how that can be achieved. Indeed the absence of labels and the propagation of moves by congruence, which are a considerable simplification giving the RL a special pleasant look, are a major drawback with respect to modularity and the definition of sensible observational semantics. Essentially, as it was remarked by many people, it seems that RL can work well only for non-structured closed systems and with the use of its support language Maude, which is well structured and enjoys a very efficient implementation.

It is worthwhile to note also that Meseguer claims that RL is an universal formalism for concurrency, since any other one can be subsumed by RL, but it is better to say that it can be coded in it in some way; this is true also for LTL, indeed in [21] we present a tricky coding of LTL transition predicates into the arrows of RL by putting, in some sense, the labels within the states. This is a technical/formal relationship, but from the point of view of the specification method it means that we have to think of the system to be specified following the LTL/SMoLCS paradigm, give an LTL specification and at the end code it in RL.

In various papers, e.g. [26, 27], and projects M. Broy has developed since 1983 an approach to the formal specification of concurrent systems which is a combination of algebraic specifications, streams, predicate logic and functional programming. The basic models are dataflow architectures and the structuring primitives are those typical for dataflows (which can be elegantly obtained as derived operators, because of the specification formalism and its semantics). However any other kinds of concurrent architectures and of communication mechanisms have to be simulated. Whenever the architecture is more or less of the dataflow style, the specifications are very elegant and convenient also for proving properties.

We now mention two approaches that only at a first sight seems to deal with the same issue, but are very different in the aims (and in the techniques). TLA, L. Lamport's Temporal Logic of Actions [45], is a very nice approach trying to deal with the various phases of development within one logical formalism, so that implementation is just logical implication. However it does not address the specification of data types and is more suited for the specification of concurrent and parallel programs and algorithms, than of large systems, where different implementation phases are needed, with much different signatures. There the notion of implementation as implication within one logic formalism cannot be exploited any more. Moreover, another major distinction is "typing" which, absolutely needed in modular development of systems, is less important in the specification of algorithms and small systems and thus absent for purpose in TLA. UNITY, by Chandy and Misra [31] offers a kind of standard programming language for describing sequential nondeterministic processes, a temporal logic for express properties on them, a way to modularly compose the specifications out of simpler ones, and a deductive system for proving properties about the specified process, also following the modular structure of the specification. Forgetting still existing problems in the formal definition of the various parts, UNITY is essentially apt to handle nondeterministic closed systems; the authors explicitly claim that they want to abstract w.r.t. the concurrent/parallel aspects of the system, e.g. as how many are the component processes and in which way they cooperate, and so on; for them such aspects must be considered only in successive implementation phases. Notice also that neither TLA nor UNITY have a notion of process as an agent, which have to be recovered by a kind of simulation. Apart from making a difference with LTL/SMoLCS, this seems to constitute a major difficulty for using TLA and UNITY with an overall object oriented approach.

Finally we close the section again emphasizing that recently some attention has been given to a complete different viewpoint, where the states of a dynamic system are modelled as algebras, which change the structure in their evolutions. Clearly, the basic idea corresponds to the "evolving algebras" of Gurevich [42]; notice however that "evolving algebras" is essentially an approach for providing operational semantics of programming languages and does not support at all abstract specifications of systems nor data. A theoretical foundation of the state-as-algebra approach, based on notion of d-oid, the extension of the notion of algebra to the dynamic one, has been developed by Astesiano and Zucca in [23, 22]; but specification issues have not yet been tackled. The issue of specification has been addressed by Dauchy and Gaudel [36]; moreover a kind of manifesto supporting the approach and building up especially on previous work in [23, 36] has been issued by Ehrig and Orejas in [39]. However the topic is not ripe yet to assess the potentialities of the state-as-algebra approach, especially in connection with the issue of concurrency (perhaps easier to tackle within the state-as-term approach).

7.2. Specifying CA using other approaches

In all cases below, the length and the complexity of the resulting specifications are comparable with the LTL/SMoLCS one; the only difference is that in some cases the text of the specification written in the chosen formalism is shorter, but relevant information have to be added apart using natural language and/or mathematical notations (e.g. also in CCS and TLA we have to describe which are the used values).

TLA Each TLA specification is about one system starting in some initial state, thus we have no way to compare two different CA's and see e.g. whether they are equivalent w.r.t. the broadcasting communications performed. As a consequence we cannot handle the higher-order features of CA.

In TLA no notion of process (component of the system) is provided, and so a CA process will be described in a specification by a set of variables recording its relevant information. Thus the creation of a new process should result in "allocating" new variables, and that cannot be done. There could be a tricky way to overcome this problem by giving a specification with an infinite number of variables and of actions (there are variables for all possible processes) plus extra boolean variables saying if a process is alive or not and creation will result in making one more process alive.

For this reason, we cannot give *one* TLA specification for CA (also if the command for creating new processes is dropped); instead we have to give *one* TLA specification for the CA with n process, for each $n \geq 1$.

Since in TLA there is no notion of type, the various data used in CA (booleans, integers, arrays, local memories, commands, ...) should be considered in the documentation when describing the values of the used variant of TLA.

The TLA specification of one CA may be structured in parts following its concurrent structure: a part for each process and one for the buffer; the connection among these parts is given by "interface variables" (i.e. variables used in more than one part). In CA we have different kinds of cooperations among the CA components; in TLA each of them is simulated by using a bunch of variables as buffers and as semaphores for handling the access to such buffers; so we have a kind of low level description of such cooperations.

Finally notice that since there are no combinators for terms representing processes or actions, there is no possibility of using structural induction.

UNITY Similar remarks and restrictions can be done when trying to use UNITY, the only difference is that instead of "actions" we have the alternatives of a guarded command.

CCS/ACP/LOTOS No way to handle the higher-order concurrent features (unless we upgrade to higher-order CCS and the like).

In general each CA process and the buffer can be described by a more or less complex behaviour in the chosen language (e.g. the local memory will be simulated by a behaviour); the data used in CA can be algebraically specified in LOTOS or described apart in the other cases (clearly we are considering only the variants of the above calculi with value passing).

The cooperation among the processes and the buffer can be simulated by using the constructs of the chosen language; clearly this part may be simple or difficult depending on the CA variant and on the chosen language: e.g. it is difficult to simulate the broadcast communication in CCS, while it can be done easily in LOTOS. Generally speaking, the cooperation mechanisms have to be simulated (implemented) by those proper of the target formalism, thus loosing abstraction compared to LTL/SMoLCS.

Rewriting Logic For what concerns Rewriting Logic (shortly RL) we can proceed in two ways:

Coding: in [21] we show how to code a subclass of LTL specifications into RL; our CA specification after changing some details (e.g. transforming the auxiliary transition relation $\rightsquigarrow\!\!\rightsquigarrow\!\!>$ into RL basic arrows on a different sort, just a copy of the system states) is in such class; so we have also an RL specification of CA. Clearly such specification has been obtained by using the ideas, techniques and methods of LTL/SMoLCS without any reference to RL; but now e.g. we can use some prototyping tools of RL on it.

Following the ideas, techniques and methods of RL: We can follows what we have done for LTL/SMoLCS; but we have to prevent the application of the closure rules (at least that of congruence); thus we have to write only the axioms for the transitions of the system states, using neither the process transitions nor those corresponding to auxiliary actions (notice that, e.g., if we define a transition for one process, such transition may fire when the process is used as a value and thus preventing to handle the higher-order features).

The specification of the interleaving variant of CA may be done in a reasonable way; but we have problems for free parallelism or for the execution policy where several accesses to the buffer may be done simultaneously; in such cases we have to write infinite rewriting rules, one for each possible combination of activities of any group of process components.

Stream processing functions A fair premise: the CA example has been chosen to illustrate the features of the LTL/SMoLCS approach and, since its structure is not easily amenable to a data flow structure, we cannot exploit the best features of stream processing functions, which is very elegant and convenient on data flow architectures.

No immediate way to handle higher-order concurrent features.

Each component (processes and buffer) can be simulated by a stream processing function and the streams can be used to connect such components and thus to simulate the cooperations among them. The problem is to control the activity in such components, e.g. in the interleaving case or when we have a mutual exclusion on some kinds of buffer access. The only way to handle this point is to put up an auxiliary controller process which schedules the activities of the components.

8. Conclusion

Looking back at our presentation two main features of our approach clearly emerge, distinguishing it from other approaches.

First, it supports full integration of the specifications of static data and dynamic elements, taking the strong view that processes are themselves data; as a consequence we get higher-order concurrency by free. The support is provided both at the design level by classical logical (algebraic) specification techniques, and at the requirement level, where models are LT-structures (particular first-order structures). Considering processes as data and thus allowing complete freedom in the specification of the communication mechanisms and cooperation policy, is a distinguishing feature w.r.t. LOTOS, for example.

Second, the specification of processes is intended to be driven by the labelled transition system model, in the sense that what we specify as concurrent behaviour are the states, the labels and the transitions. In this respect we follow strictly CCS, where first the transition semantics is defined [49]; indeed as a most significant example, the rules of transition semantics of CCS are an LT-specification for CCS. To mention a different viewpoint, we can take the classical ACP specifications (see [24]) where the algebraic laws of the various operations are defined, but the labelled transition model remains in the background; this amounts to take the CCS algebraic laws, which are theorems, and turn them into axioms, which then encode also some form of observational semantics. We have found in many experiences that the transition driven approach, quite operational, is well suited to the system engineer intuition, while the algebraic laws defining the operations are suitable and nice either when referring to a single language into which the specifications are coded, or when the laws are intuitive, like the commutativity and associativity of the parallel operator and of the nondeterministic choice of CCS.

Third, the transitions are expressed by predicates and thus are completely within first-order logic, contrary, e.g., to Rewriting Logic, where the transition arrow is at the meta-level.

The outlined approach has been developed since 1984 mainly by the authors, (see [13, 11, 57]) with an important contribution of M. Wirsing in the early stages (see [6]); later on other people contributed to some related aspects, notably A. Giovini († 93) for the observational semantics [4] and F. Morando for the development of a rapid prototyping system [40, 8]. The idea of conditional LTL was the core of [6], where its specifications were called algebraic transition systems; the concept of higher-order concurrency was explored in [13] and its semantics foundations laid down in [2] and generalized in [4]. Requirement LT-specifications have been dealt with especially in [34, 35, 59, 18] and some methodological aspects have been addressed in [19].

A notable feature of our approach is that its development has been motivated throughout the various phases by some concrete experimental problems. The original approach addressing the design phases was developed in connection with and applied to the specification of a prototype Campus Net, in the project CNET [7]. Then the approach was extended to deal with semantic issues of languages with concurrent features and applied to give the first complete formal definition of Ada, within an EEC-MAP project (see e.g. [12, 1, 9]). Finally, in a four years long project partially supported by ENEL (the National Electricity Company of Italy), two test cases have been addressed, which have been influential for the requirement phase and methodological aspects, discussed jointly with the people on the industry side. The case studies concern the specification of a hydro-electric system [60] and of a high-voltage station for the distribution of the electric power [61].

Looking back at our ten-years experience we cannot deny the general difficulty of putting formal methods into practice. We are now convinced that it is essential to have at hand a semiformal approach, where corresponding semiformal and formal specifications proceed in parallel, following what we have called a “two-rail approach” [19]. Moreover, but that is already well-known, software tools are essential in all phases, from editing to verification. What we have at hand in our method is still insufficient; in particular we are now looking for tools also allowing graphical representations (which however cannot replace the textual ones, especially for big specifications).

There is also another more basic research aspect coming out of our experience, which has to deal with metaformalism.

Each of the papers cited at the end of Sect. 5.2 presents a particular feature to be added to the basic LTL (as temporal/deontic combinators, abstract events, entities); these features are mutually consistent, i.e. we can have a complete formalism including all of them. However in most applications only few of them are needed (e.g. the industrial case study of [61] requires an entity specification with temporal logic, while that of [60] only the temporal logic features). In some other cases, instead, it is sufficient to incorporate the simpler linear-time temporal logic combinators, or, when many properties are about failures and fault-tolerance, deontic-logic combinators. Moreover, here we have presented many-sorted total LTL, i.e. where each LT-structure is a particular total many-sorted first-order structure, while for some other applications partial/order-sorted structures are more adequate (depending on the features of the static parts).

The above considerations make clear that we may have a whole family of LTL's, where each instance is appropriate for some kind of applications; so using the fact that all of them are institutions, in [30, 29] we have tried to develop appropriate operations having institutions as arguments and results (each one more or less corresponds to add a feature, as temporal-logic combinators, entities, ...) for being able to modularly define the right variant of LTL we need, and then to use it for the specification of dynamic elements.

Acknowledgement. We want to acknowledge the role of many people in the development of the presented approach, starting with Martin Wirsing, and then the other members of the Genova group involved in the CNet and in the Ada projects (especially Alessandro Giovini, Elena Zucca and Franco Mazzanti), Franco Morando for the toolset, and the ENEL people cooperating in the development of the cases studies: Ernani Crivelli and Valeria Filippi.

A. Logical/algebraic specifications

A.1. Basic notions

Definition 8. *A* (many-sorted) first-order signature *(shortly, a* signature*) is a triple $\Sigma = (S, OP, PR)$, where*

- *S is a set (the set of the* sorts*);*
- *OP is a family of sets: $\{OP_{w,s}\}_{w \in S^*, s \in S}$; $Op \in OP_{w,s}$ is an* operation symbol *(of* arity *w and* target *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 $Op: s_1 \times \ldots \times s_n \to s$ for $Op \in OP_{s_1 \ldots s_n, s}$, $Pr: s_1 \times \ldots \times s_n$ for $Pr \in PR_{s_1 \ldots s_n}$.

Definition 9. *A Σ-*first-order structure *(shortly a Σ-*structure*) is a triple*

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

consisting of the carriers*, 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 $Op: s_1 \times \ldots \times s_n \to s$, then $Op^{\mathrm{A}}: \mathrm{A}_{s_1} \times \ldots \times \mathrm{A}_{s_n} \to \mathrm{A}_s$ is a (total) function;*
- *if $Pr: s_1 \times \ldots \times s_n$, then $Pr^{\mathrm{A}} \subseteq \mathrm{A}_{s_1} \times \ldots \times \mathrm{A}_{s_n}$.*
  Usually we write $Pr^{\mathrm{A}}(a_1, \ldots, a_n)$ instead of $(a_1, \ldots, a_n) \in Pr^{\mathrm{A}}$. □

Definition 10. *Given a signature Σ and an S-indexed family of sets of variables X, the* term structure *on Σ and X, $T_\Sigma(X)$, is the Σ-structure 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-structure, $t \in T_\Sigma(X)$ and $V: X \to \mathrm{A}$ is a variable evaluation*, i.e. a sort-respecting assignment of values in A to* all *the variables in X, then the* interpretation of t in A w.r.t. V*, denoted by $t^{\mathrm{A},V}$, is defined as usual; if t is a ground term, then we use the notation t^{A}.* □

In this paper we assume that structures have *nonempty carriers* (as this applies to term structures as well, thus we have an implicit assumption on signatures: that they contain “enough constants symbols”).

Definition 11. *Let A be a Σ-structure.*

- *If $\Sigma' = (S', OP', PR')$ is a subsignature of Σ, then $\mathrm{A}_{|\Sigma'}$ denotes the Σ'-structure B defined as follows:*
  - *$\mathrm{B}_s = \mathrm{A}_s$ for all $s \in S'$,*

  - $Op^{\mathrm{B}} = Op^{\mathrm{A}}$ *for all* $Op \in OP'$,
  - $Pr^{\mathrm{B}} = Pr^{\mathrm{A}}$ *for all* $Pr \in PR'$.
- *If* C *is a set of operation symbols of* Σ, *then* A *is* C-generated *iff for all* $s \in Rsorts(C)$, *for all* $a \in \mathrm{A}_s$, *there exists* X *a family of variables s.t.* $X_{s'} = \emptyset$ *for all* $s' \in Rsorts(C)$, *a variable evaluation* $V\colon X \to \mathrm{A}$ *and* $t \in T_{\Sigma(C)}(X)_s$ *s.t.* $t^{V,\mathrm{A}} = a$; *where:*
  - $Rsorts(C) = \{s \mid$ *there exists* $Op \in C$ *with result sort* $s\}$,
  - $Sorts(C) =$
    $\{s \mid$ *there exists* $Op \in C$ *having* s *either as argument or as result sort* $\}$,
  - $\Sigma(C)$ *is the signature* $(Sorts(C), C, \emptyset)$ $(\subseteq \Sigma)$.

  If A *is* OP*-generated, then it is said* term-generated*; in such cases for all* $s \in S$, $a \in \mathrm{A}_s$ *there exists* $t \in (T_\Sigma)_s$ *s.t.* $a = t^{\mathrm{A}}$. □

Definition 12. *If* A *and* B *are* Σ*-structures, a* homomorphism h *from* A *into* B *(written* $h\colon \mathrm{A} \to \mathrm{B}$*) is a family of* total *functions* $h = \{h_s\}_{s \in S}$ *where for all* $s \in S$ $h_s\colon \mathrm{A}_s \to \mathrm{B}_s$ *and*

- *for all* $Op \in OP$ $h_s(Op^{\mathrm{A}}(a_1, \ldots, a_n)) = Op^{\mathrm{B}}(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$;
- *for all* $Pr \in PR$: *if* $Pr^{\mathrm{A}}(a_1, \ldots, a_n)$, *then* $Pr^{\mathrm{B}}(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$. □

Σ-structures and homomorphisms form a category.

First-order logic and the interpretation of its formulae on a Σ-structure A is defined as usual. We write $\mathrm{A}, V \models \theta$ when the interpretation of the formula θ in A w.r.t. V yields true; then θ is *valid* in A (written $\mathrm{A} \models \theta$) whenever $\mathrm{A}, V \models \theta$ for all evaluations V.

A Σ-structure A is a *model* of a set of formulae AX on Σ iff for all $\theta \in AX$ $\mathrm{A} \models \theta$.

A *conditional formula* is a first-order formula on Σ and X having form $\wedge_{i=1,\ldots,n} \alpha_i \supset \alpha$, where α_i, α are *atoms*, and an atom has form either $t_1 = t_2$ or $Pr(t_1, \ldots, t_n)$, with Pr predicate symbol and $t_i \in T_\Sigma(X)$.

Definition 13. *The Birkhoff deductive system for a set of conditional formulae* CAX, *sound and complete w.r.t. the models of* CAX, *consists of the axioms* CAX *and of the following rules.*

$$REF \quad \frac{}{t = t} \qquad SYM \quad \frac{t = t'}{t' = t} \qquad TRANS \quad \frac{t = t' \quad t' = t''}{t = t''}$$

$$CONG1 \quad \frac{t_1 = t'_1 \ \ldots \ t_n = t'_n}{Op(t_1, \ldots, t_n) = Op(t'_1, \ldots, t'_n)}$$

$$CONG2 \quad \frac{Pr(t_1, \ldots, t_n) \quad t_1 = t'_1 \ \ldots \ t_n = t'_n}{Pr(t'_1, \ldots, t'_n)}$$

$$MP \quad \frac{\wedge_{i=1,\ldots,n} \ \alpha_i \supset \alpha \qquad \alpha_1 \ldots \alpha_n}{\alpha} \qquad SUB \quad \frac{\theta}{\theta[t/x]}$$

If θ *can be proved using the Birkhoff system for* CAX, *then we write* $CAX \vdash \theta$. □

Definition 14. *Let $\mathcal{C}$ be a class of Σ-structures.* $\mathrm{A} \in \mathcal{C}$ *is said* initial *in $\mathcal{C}$ if for every $\mathrm{A}' \in \mathcal{C}$ there exists one and only one homomorphism $\phi\colon \mathrm{A} \to \mathrm{A}'$.* □

Proposition 3.

- *If* I *is initial in $\mathcal{C}$, then:*

  for all terms t, t', $\mathrm{I} \models t = t'$ iff $\mathrm{A} \models t = t'$ for all $\mathrm{A} \in \mathcal{C}$
  for all predicates Pr in Σ and all terms $t_1, \ldots, t_n$,
  $\mathrm{I} \models Pr(t_1, \ldots, t_n)$ *iff* $\mathrm{A} \models Pr(t_1, \ldots, t_n)$ *for all $\mathrm{A} \in \mathcal{C}$.*

- *If CAX is a set of conditional formulae, then there exists* I *initial in the class of the models of CAX characterized by:*

  for all terms t, t', $\mathrm{I} \models t = t'$ iff $CAX \vdash t = t'$
  for all predicates Pr in Σ and all terms $t_1, \ldots, t_n$,
  $\mathrm{I} \models Pr(t_1, \ldots, t_n)$ *iff* $CAX \vdash Pr(t_1, \ldots, t_n)$. □

A.2. Predefined parametric specifications

Finite multisets

```
spec  MSET(ELEM, elem) =
      enrich ELEM by
      sorts    mset
      opns     ∅: → mset
               {_}: elem → mset
               _ | _: mset × mset → mset
      axioms
            ms | ∅ = ms
            ms₁ | ms₂ = ms₂ | ms₁
            (ms₁ | ms₂) | ms₃ = ms₁ | (ms₂ | ms₃)
```

Usually a term of the form $\{t_1\} \mid \ldots \mid \{t_n\}$ is simply written $t_1 \mid \ldots \mid t_n$.

Finite maps

spec MAP(INDEX, $index$, Eq, ELEM, $elem$, $Error$) =
    enrich ELEM + INDEX **by**
    sorts map
    opns $[] \colon \to map$
        $\_[\_/\_] \colon map \times elem \times index \to map$
        $\_(\_) \colon map \times index \to elem$
    axioms
        $[](i) = Error$
        $m[e/i](i) = e$
        $i\ Eq\ i' = False \supset m[e/i](i') = m(i')$

10

1. E. Astesiano, A. Giovini, F. Mazzanti, G. Reggio, and E. Zucca. The Ada Challenge for New Formal Semantic Techniques. In *Ada: Managing the Transition, Proc. of the Ada-Europe International Conference, Edimburgh, 1986,*, pages 239–248. University Press, Cambridge, 1986.
2. E. Astesiano, A. Giovini, and G. Reggio. Generalized Bisimulation in Relational Specifications. In *Proc. STACS'88 (Symposium on Theoretical Aspects of Computer Science)*, number 294 in Lecture Notes in Computer Science, pages 207–226. Springer Verlag, Berlin, 1988.
3. E. Astesiano, A. Giovini, and G. Reggio. Processes as Data Types: Observational Semantics and Logic. In I. Guessarian, editor, *Proc. of 18-eme Ecole de Printemps en Informatique Theorique, Semantique du Parallelism*, number 469 in Lecture Notes in Computer Science, pages 1–20. Springer Verlag, Berlin, 1990.
4. E. Astesiano, A. Giovini, and G. Reggio. Observational Structures and their Logic. *T.C.S.*, 96:249–283, 1992.
5. E. Astesiano, A. Giovini, G. Reggio, and E. Zucca. An Integrated Algebraic Approach to the Specification of Data Types, Processes and Objects. In M. Wirsing and J.A. Bergstra, editors, *Algebraic Methods: Theory, Tool and Applications*, number 394 in Lecture Notes in Computer Science, pages 91–116. Springer Verlag, Berlin, 1989.
6. 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.
7. E. Astesiano, F. Mazzanti, G. Reggio, and E. Zucca. Formal Specification of a Concurrent Architecture in a Real Project. In *A Broad Perspective of Current Developments, Proc. ICS'85 (ACM International Computing Symposium)*, pages 185–195. North-Holland, Amsterdam, 1985.
8. E. Astesiano, F. Morando, and G. Reggio. The SMoLCS Toolset. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Proc. of TAPSOFT '95*, number 915 in Lecture Notes in Computer Science, pages 810–811. Springer Verlag, Berlin, 1995.
9. E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini, P. Inverardi, E. Karlsen, F. Mazzanti, J. Storbank Pedersen, G. Reggio, and E. Zucca. The Draft Formal Definition of Ada. Deliverable, CEC MAP project: The Draft Formal Definition of ANSI/STD 1815A Ada, 1986.
10. E. Astesiano and G. Reggio. On the Specification of the Firing Squad Problem. In *Proc. of the Workshop on The Analysis of Concurrent Systems, Cambridge, 1983*, number 207 in Lecture Notes in Computer Science, pages 137–156. Springer Verlag, Berlin, 1985.
11. E. Astesiano and G. Reggio. An Outline of the SMoLCS Approach. In M. Venturini Zilli, editor, *Mathematical Models for the Semantics of Parallelism, Proc. Advanced School on Mathematical Models of Parallelism, Roma, 1986*, number 280 in Lecture Notes in Computer Science, pages 81–113. Springer Verlag, Berlin, 1987.
12. E. Astesiano and G. Reggio. Direct Semantics of Concurrent Languages in the SMoLCS Approach. *IBM Journal of Research and Development*, 31(5):512–534, 1987.

13. 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.
14. E. Astesiano and G. Reggio. The SMoLCS Approach to the Formal Semantics of Programming Languages – A Tutorial Introduction. In A.N. Habermann and U. Montanari, editors, *System Development and Ada, Proc. of CRAI Workshop on Software Factories and Ada, Capri 1986*, number 275 in Lecture Notes in Computer Science, pages 81–116. Springer Verlag, Berlin, 1987.
15. E. Astesiano and G. Reggio. A Structural Approach to the Formal Modelization and Specification of Concurrent Systems. Technical Report PDISI-92-01, DISI, Università di Genova, Italy, 1992.
16. E. Astesiano and G. Reggio. A Metalanguage for the Formal Requirement Specification of Reactive Systems. In J.C.P. Woodcock and P.G. Larsen, editors, *Proc. FME'93: Industrial-Strength Formal Methods*, number 670 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1993.
17. E. Astesiano and G. Reggio. Algebraic Specification of Concurrency (invited lecture). In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, number 655 in Lecture Notes in Computer Science, pages 1–39. Springer Verlag, Berlin, 1993.
18. E. Astesiano and G. Reggio. Specifying Reactive Systems by Abstract Events. In *Proc. of Seventh International Workshop on Software Specification and Design (IWSSD-7)*. IEEE Computer Society, Los Alamitos, CA, 1993.
19. E. Astesiano and G. Reggio. Formally-Driven Friendly Specifications of Concurrent Systems: A Two-Rail Approach. Technical Report DISI–TR–94–20, DISI – Università di Genova, Italy, 1994. Presented at ICSE'17-Workshop on Formal Methods, Seattle April 1995.
20. E. Astesiano and G. Reggio. A Dynamic Specification of the RPC-Memory Problem. Lecture Notes in Computer Science. Springer Verlag, Berlin, 1996. To appear.
21. E. Astesiano and G. Reggio. On the Relationship between Labelled Transition Logic and Rewriting Logic. Technical Report DISI–TR–96–19, DISI – Università di Genova, Italy, 1996.
22. E. Astesiano and E. Zucca. D-oids: a Model for Dynamic Data Types. *Mathematical Structures in Computer Science*, 5(2):257–282, 1995.
23. E. Astesiano and E. Zucca. A Free Construction of Dynamic Terms. *Journal of Computer and System Sciences*, 52(1):143–156, 1996.
24. J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, Cambridge, 1990.
25. M. Bettaz and G. Reggio. A SMoLCS Based Kit for Defining the Semantics of High-Level Algebraic Petri Nets. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification*, number 785 in Lecture Notes in Computer Science, pages 98–112. Springer-Verlag, Berlin, 1994.
26. M. Broy. Specification and Top Down Design of Distributed 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 4–28. Springer Verlag, Berlin, 1985.
27. M. Broy. Predicative Specifications for Functional Programs Describing Communicating Networks. *Information Processing Letters*, 25:2, 1987.
28. M. Broy and M. Wirsing. Partial Abstract Types. *Acta Informatica*, 18:47–64, 1982.
29. M. Cerioli and G. Reggio. Algebraic Oriented Institutions. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology (AMAST'93)*, Workshops in Computing. Springer Verlag, London, 1993.
30. M. Cerioli and G. Reggio. Institutions for Very Abstract Specifications. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification*, number 785 in Lecture Notes in Computer Science, pages 113–127. Springer-Verlag, Berlin, 1994.
31. M. Chandy and J. Misra. *Parallel Program Design: a Foundation*. Addison-Wesley, 1988.
32. X-J. Chen and C. Montangero. Compositional Refinements in Multiple Blackboard Systems. *Acta-Informatica*, 32(5):459–476, 1995. A short version appeared in *Proceeding of ESOP'92*, Lecture Notes in Computer Science n. 582, Springer, 1992.

33. E. Coscia and G. Reggio. Deontic Concepts in the Algebraic Specification of Dynamic Systems: The Permission Case. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specification*, number 1130 in Lecture Notes in Computer Science, pages 161–182. Springer Verlag, Berlin, 1996. 11th Workshop on Specification of Abstract Data Types joint with the 8th general COMPASS workshop. Oslo, Norway, September 1995. Selected papers.
34. 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.
35. G. Costa and G. Reggio. Specification of Abstract Dynamic DataTypes: A Temporal Logic Approach. *T.C.S.*, 173, 1997. To appear.
36. P. Dauchy and M.C. Gaudel. Implicit State in Algebraic Specifications. In U. W. Lipeck and G. Koschorreck, editors, *Proc. International Workshop Is-Core'93, Hannover September 1993*, 1993.
37. H.D. Ehrich. On the Realization and Implementation. In *Proc. MFCS'81*, number 118 in Lecture Notes in Computer Science, pages 271–280. Springer Verlag, Berlin, 1981.
38. H. Ehrig, H.J. Kreowski, B. Mahr, and P. Padawitz. Algebraic Implementation of Abstract Data Types. *T.C.S.*, 20:209–263, 1982.
39. H. Ehrig and F. Orejas. Dynamic Abstract Data Types: An Informal Proposal. *Bulletin of the EATCS*, (53), 1994.
40. A. Giovini, F. Morando, and A. Capani. Implementation of a Toolset for Prototyping Algebraic Specifications of Concurrent Systems. In *Proc. ALP'92*, number 632 in Lecture Notes in Computer Science, pages 335–349. Springer Verlag, Berlin, 1992.
41. J. Goguen and J. Meseguer. Models and Equality for Logic Programming. In *Proc. TAPSOFT'87, Vol. 2*, number 250 in Lecture Notes in Computer Science, pages 1–22. Springer Verlag, Berlin, 1987.
42. Y. Gurevich. Evolving Algebras, a Tutorial Introduction. *Bulletin of the EATCS*, (43):264–284, 1991.
43. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, London, 1985.
44. I.S.O. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS 8807, International Organization for Standardization, 1989.
45. L. Lamport. The Temporal Logic of Actions. Technical Report 79, Digital, Systems Research Center, Palo Alto, California, 1991.
46. Wei Li. An Operational Semantics of Multitasking and Exception Handling in Ada. In *Proc. AC Ada Tech. and Tutorial Conference.* ACM Press, 1982.
47. 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.
48. J. Meseguer. Conditional Rewriting as a Unified Model of Concurrency. *T.C.S.*, 96:73–155, 1992.
49. R. Milner. *A Calculus of Communicating Systems.* Number 92 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1980.
50. R. Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.
51. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes - Part I. Technical Report ECS-LFCS-89-85, LFCS-University of Edinburgh, 1989.
52. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes - Part II. Technical Report ECS-LFCS-89-86, LFCS-University of Edinburgh, 1989.
53. R. De Nicola and M. Hennessy. Testing Equivalence for Processes. *T.C.S.*, 34:181–205, 1984.
54. D. Park. Concurrency and Automata on Infinite Sequences. In *Proc. 5th GI Conference*, number 104 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1981.
55. G. Plotkin. A Structural Approach to Operational Semantics. Lecture notes, Aarhus University, 1981.
56. 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.
57. G. Reggio. *Una Metodologia per la Specifica di Sistemi e Linguaggi Concorrenti.* Ph. D. Thesis, Università di Genova-Pisa-Udine, 1988. In italian.

58. G. Reggio. Entities: an Institution 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.
59. G. Reggio. Event Logic for Specifying Abstract Dynamic Data Types. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, number 655 in Lecture Notes in Computer Science, pages 292–309. Springer Verlag, Berlin, 1993.
60. G. Reggio and E. Crivelli. Specification of a Hydroelectric Power Station: Revised Tool-Checked Version. Technical Report DISI–TR–94–17, DISI – Università di Genova, Italy, 1994.
61. G. Reggio, A. Morgavi, and V. Filippi. Specification of a High-Voltage Substation. Technical Report PDISI-92-12, DISI – Università di Genova, Italy, 1992.
62. W. Reisig. Petri Nets and Algebraic Specifications. *T.C.S.*, 80:1–34, 1991.
63. G. Winskel and M. Nielsen. Models for Concurrency. Technical Report 492, DAIMI PB, 1992.
64. M. Wirsing. Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoret. Comput. Sci.*, volume B, pages 675–788. Elsevier, 1990.

This article was processed by the author using the LaTeX style file *cljour1* from Springer-Verlag.