Acta Informatica manuscript No.
(will be inserted by the editor)

Labelled Transition Logic: An Outline*

E. Astesiano and G. Reggio

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Via Dodecaneso, 35 – Genova 16146 – Italy
Fax ++ 39 - 010 - 3536699
{ astes,reggio } @ disi.unige.it

The date of receipt and acceptance will be inserted by the editor

■

Abstract. 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/parallel/distributed 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, applications and documentation.

The overall main feature of the method is its logical (algebraic) character, because it extends to labelled transition systems the logical/algebraic specification method of abstract data types; moreover systems 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 systems, 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 systems, 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.

---

* Work partially supported by CoFI (ESPRIT Working Group 29432) and by MURST - Programma di Ricerca Scientifica di Rilevante Interesse Nazionale SALADIN.

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, because most software systems are dynamic, in the sense that they deal with structures evolving in time and frequently also including some form of parallelism, concurrency, reactivity and distribution. Thus (a little) later on and extensively during the 80's, the modelling 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 dynamic systems (processes) 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 dynamic systems as dynamic data, that is as elements of special sorts in a first-order structure. As in CCS and other approaches, they are are modelled as labelled transition systems (i.e., nondeterministic automata with labelled transitions). However, to keep the specification at the abstract level appropriate for the system to be specified, both states and labels are in general data of a suitable sort.

To support this view we adopt as basic mathematical models what we call "LT-structures", that are many-sorted first-order structures where some special sorts (called dynamic), representing systems (or better system states), have both an associate label sort and a ternary transition predicate, and so correspond to labelled transition systems.

Notice that, because of the associate labelled transition system, an element of a dynamic sort representing a state in an LT-structure also represents a whole system, 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 on the integration of concurrency and algebraic specifications also of interest and receiving some attention is the one driven by the "state-as-algebra" idea (see [1] also for references).

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

Because of the view that dynamic systems 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 specifications, with several advantages, which constitute distinguishing features of our approach.

Static and dynamic properties of systems The LTL formulae express both *static* and *dynamic* properties of systems. Static means properties about the static structure of the systems; for example, 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 systems (properties about the transition predicate); for example, 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 the static properties help also express the dynamic ones in a simpler mode; for example, the unique axiom

$$b_1 \xrightarrow{l} b_1' \;\Rightarrow\; b_1 + b_2 \xrightarrow{l} b_1'$$

completely characterizes from a behavioural point of view the CCS nondeterministic choice $+$, because $+$ is commutative (by the static properties).

Integration of abstract data type specifications and system specifications Because the LTL formulae include those of the first-order logic with equality, we can use LTL also to express the properties of the data used by the systems (e.g., values, messages, ...); thus we can integrate the classical specifications of abstract data types with axiomatic specifications of systems. Moreover, because the systems themselves are data, we can describe data types with system components (e.g., functions from or into systems), systems with different types of dynamic components and complex systems where the dynamic components of some type are composed out of systems of other types.

Extension to concurrency of techniques/results of algebraic specification We can extend all well-known results, techniques and methodological principles developed in the field of the algebraic specification of abstract data types to LT-specifications (and so to the specification of concurrency). As an outstanding example, we can model the distinction between requirement and design specifications, with the associated notion of implementation, following the formalization given in the context of abstract data type specifications, as loose and initial specifications, with a very general notion of implementation.

Roughly speaking a *requirement specification* defines general requirements on a system for some level of abstraction; a bit more precisely it requires a 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 that may differ even for some substantial aspect).

A *design specification* is meant to describe exactly a system, for some level of abstraction; the idea is to determine a system as the one that satisfies all and only the properties listed in the specification and their logical consequences. Here an important constraint concerns the fact that we have to formulate the properties in a design specification 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., [65]): 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 system, 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 [4, 5]).

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).

Summary

The paper gives a general outline of the fundamental aspects of an approach to specification that has been developed since 1984 also benefiting from the experience gained in some significant projects and industrial test cases, as reported in the conclusion. The approach consists by now of theoretical foundations, the formalism LTL, and the companion methodological software engineering guidelines (see [18, 21]) 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.

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. For a variety of more relevant applications of LTL/SMoLCS, see:

- specifications of case studies, as in [8, 11, 19, 60, 61];
- its usage as a base for formally defining other formalisms, e.g., [24], for formalizing in a modular way various kinds of higher-order algebraic Petri nets, [31] for a variant of extended shared Prolog, [6] for object-oriented features, and, more recently, [59] for parts of UML;
- for defining the semantics of concurrent/parallel programming languages, as Ada, [2, 10, 13, 15].

Recently, LTL has been chosen as the basic underlying formalism for defining an extension, called Casl-Ltl [58], of the algebraic specification language Casl [63], to deal with reactive, concurrent and parallel systems. Casl ("Common Algebraic Specification Language") is the language developed within CoFI (the European "Common Framework Initiative", [51]) for the algebraic specification of software and systems, partially supported by the EU ESPRIT program, which brings together research institutions from all over Europe. Casl was intended to set a standard unifying the various approaches to algebraic specification and specification of abstract data types.

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 [54] for the use of labelled transition systems and their operational specifications, CCS [47, 48] for the concept of process as labelled transition tree with various observational semantics, and finally the Munich algebraic approach (see, e.g., [27]) for a first example of an algebraic version of CCS.

In Appendix A we have reported the basic notations, definitions and results about the first-order specifications used in the paper.

2 Models of Dynamic Systems

In this paper we will use the following terminology.

The word *system* denotes a dynamic system of whatever kind, and so evolving along the time, without any assumption about other aspects of its behaviour; thus a system may be a communicating/ nondeterministic/sequential/... process, a reactive/parallel/concurrent /distributed/... system, a software architecture, but also an object-oriented system (a community of interacting objects), and an agent or an agent system.

The words *structured system* denote a system consisting of different components (at least two) that are themselves systems cooperating among them; conversely a *simple system* is a system without components. Then we classify the components of a structured system into *active* and *passive*; the latter are those that cannot act by themselves, but perform some activity only as the result of some action of some other active component.

The CS concurrent system, see Sect. 4.1, and the Ada programs, which have as components the tasks, see [10], are examples of structured systems; the CS processes and the task components of Ada programs are active components; instead the buffer of the CS system and the shared memory of Ada programs are passive components.

The picture in Fig. 1 presents our classification and related terminology as an object-oriented class diagram.

2.1 Labelled transition systems for modelling systems

For modelling systems we adopt the well-known and accepted technique that consists in viewing a system as a labelled transition tree defined by a labelled transition system (see [47] and [55]).

Definition 1. *A* labelled transition system *(shortly* lts*) is a triple* $(STATE, LABEL, \rightarrow)$*, where* $STATE$ *and* $LABEL$ *are sets, 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 is usually denoted by* $s \xrightarrow{l} s'$*.* □

A system S may be represented by an lts

$$(STATE, LABEL, \rightarrow)$$

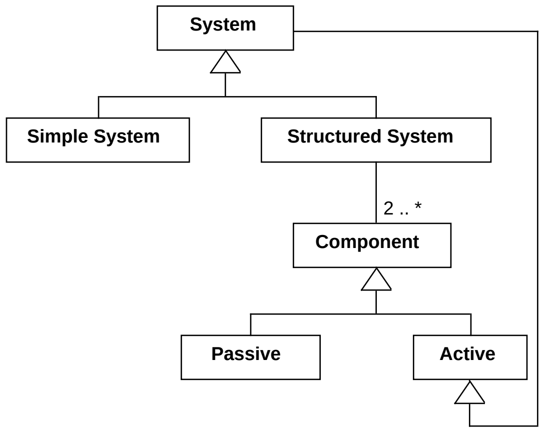

Fig. 1 The used terminology

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 S and the transition relation $\rightarrow$ the possibilities of S 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: S in the state s has the *capability* of passing into the state s' by performing a transition whose interaction with the external (to S) 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, the 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 process components of CS (a concurrent system, see Sect. 4.1); using our terminology they are simple systems. We may model such processes with an lts whose states are pairs of the form $\langle c, lm \rangle$, where c is a command and lm a local memory state, and whose labels include TAU (in this paper, following the convention adopted by Milner for CCS, we use "TAU" to label internal transitions); $SEND(v, ch)$ and $REC(v, ch)$ (sending and receiving a value v through a channel ch by handshaking communication).

$$\begin{tikzcd}
\langle x := \mathit{0}; send(x, ch_{\mathit{0}}), [] \rangle \arrow[d, "TAU"] \\
\langle send(x, ch_{\mathit{0}}), [][\mathit{0}/x] \rangle \arrow[d, "SEND(\mathit{0}, ch_{\mathit{0}})"] \\
\langle skip, [][\mathit{0}/x] \rangle
\end{tikzcd}$$

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

$$\begin{tikzcd}
& \langle rec(x, ch_{\mathit{0}}), [] \rangle \arrow[dl, "REC(\mathit{0}, ch_{\mathit{0}})"'] \arrow[dr, "REC(\mathit{1}, ch_{\mathit{0}})"] & & \cdots \\
\langle skip, [][\mathit{0}/x] \rangle & & \langle skip, [][\mathit{1}/x] \rangle & \cdots
\end{tikzcd}$$

Fig. 3 Transition tree associated with a process performing an input command

$$\begin{tikzcd}
& \langle send(\mathit{2}, ch_{\mathit{1}}) + send(\mathit{3}, ch_{\mathit{2}}), [] \rangle \arrow[dl, "SEND(\mathit{2}, ch_{\mathit{1}})"'] \arrow[dr, "SEND(\mathit{3}, ch_{\mathit{2}})"] & \\
\langle skip, [] \rangle & & \langle skip, [] \rangle
\end{tikzcd}$$

Fig. 4 Transition tree associated with a process performing a nondeterministic choice

A process with command part $x := \mathit{0}; send(x, ch_{\mathit{0}})$ and empty local memory ($[]$) is modelled by the transition tree reported in Fig. 2; the first transition labelled by TAU is internal, whereas the second is a capability that will become effective only when the process is put in parallel with another one ready to receive the value 0 on channel ch_{0} ($[][\mathit{0}/x]$ denotes the local memory state where 0 is associated with x).

The tree in Fig. 3 represents a process performing an input command on channel ch_{0}; it has infinite capabilities, because it can receive any possible value; the transition labelled by $REC(n', ch_{\mathit{0}})$ will become effective only when the process is in parallel with another one ready to send the value n' on channel ch_{0}.

The tree in Fig. 4 represents a process performing the nondeterministic choice between two output commands.

We model also the structured systems (i.e., systems with components that in turn are systems) by using special lts's that are built by putting together (several) other lts's, whose transitions describe the activity of the component systems.

As an example of structured system let us consider the concurrent system CS (see Sect. 4.1), whose active components are the processes and whose passive component is the buffer. 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 CS processes and bf is a value representing the state of the shared buffer. The transitions of the process components and the buffer state determine the transitions of such lts. For instance,

if $p_1 \xrightarrow{SEND(n',ch')} p_1'$ and $p_2 \xrightarrow{REC(n',ch')} p_2'$, 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).

If $p \xrightarrow{READ(n')} p'$ and n' is the first value available in the buffer in state 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 n' has been dropped from (reading the buffer).

By associating with a system S the transition tree having root S we give an operational semantics: two systems are operationally equivalent whenever the associated transition trees are the same, see [47]. However in most cases such semantics is too fine, because it takes into account all details of the system activity. It may happen that two systems that we consider semantically equivalent have associated different transition trees. A simple case is when we consider the trees associated with two sequential CS 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. 5) are unary trees, with all arcs labelled by TAU. If we consider an input-output semantics, 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; we do not consider the differences concerning other aspects (intermediate states, number of the intermediate transitions, etc.).

$$p \xrightarrow{TAU} p_1 \xrightarrow{TAU} \quad \dots \quad \xrightarrow{TAU} p_n \xrightarrow{TAU} p_{\mathrm{F}}$$

$$\begin{array}{l} p' \xrightarrow{TAU} p'_1 \xrightarrow{TAU} \quad \dots \quad \xrightarrow{TAU} p'_n \xrightarrow{TAU} p'_{n+1} \xrightarrow{TAU} \quad \dots \\ \qquad\qquad\qquad\qquad \xrightarrow{TAU} p'_m \xrightarrow{TAU} p'_{\mathrm{F}} \end{array}$$

Fig. 5 Transition trees associated with two sequential processes

From this simple example, we understand also that we can get various interesting semantics on systems modelled by lts's depending on what we observe (see, e.g., [47, 52]). For instance, consider the well-known strong bisimulation semantics of Park [53] and Milner [47] and the trace semantics [41]. In the first case, two systems are equivalent iff they have the same associated transition trees after the states have been forgotten. In the second case, two systems 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 systems depends on what we are interested to observe: i.e., the semantics of systems is observational, in a loose sense for the moment.

2.2 LT-structures

We can represent an lts 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

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

(which corresponds to the transition relation) see, [27, 7]. The triple

$$(\mathrm{A}_{state}, \mathrm{A}_{label}, \rightarrow^{\mathrm{A}})$$

is the corresponding lts. Clearly, we can represent in this way also lts's modelling structured systems; those are 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);*

– *$DS \subseteq S$ (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 $Ds\_Lab \in S - DS$ (the sort of the labels) and a predicate $\_\_ \xrightarrow{\_\_} \_\_ : Ds \times Ds\_Lab \times Ds \in PR$ (the transition predicate).*

Given two LT-signatures, say $LT\Sigma = (\Sigma, DS)$ and $LT\Sigma' = (\Sigma', DS')$, a LT-signature morphism *$\sigma: LT\Sigma \to LT\Sigma'$ is a morphism of first-order signatures $\sigma: \Sigma \to \Sigma'$ s.t. for all $Ds \in DS$ $\sigma(Ds) \in DS'$, $\sigma(Ds\_Lab) = \sigma(Ds)\_Lab$ and $\sigma(\_\_ \xrightarrow{\_\_} \_\_ : Ds \times Ds\_Lab \times Ds) = \_\_ \xrightarrow{\_\_} \_\_ : \sigma(Ds) \times \sigma(Ds)\_Lab \times \sigma(Ds)$ (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.* □

Because 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 systems themselves are a data type (each system is given as its initial state); thus we can define systems that can be exchanged as values by other systems, systems and data having systems as components (e.g., functions from some data into systems) may be stored in memories, and so on. Hence the systems we model by LT-structures are potentially higher-order, in a sense first introduced by the authors in [14] and now well-known because of underlying famous calculi like π-calculus [49, 50] 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 [10]): the denotation of a task type is a function associating with some parameters the system 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.

Because LT-structures are particular first-order structures, we take as homomorphisms the usual homomorphisms between such structures (see Appendix A) 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 $h: \mathrm{L} \to \mathrm{L}'$) is a homomorphism from* L *into* L′ *considered as Σ-first-order structures.* □

Because LT-homomorphisms preserve the truth of predicates, they preserve also the activity of the systems; thus, 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., [64]) 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,* 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). Then:*

- $\mathrm{LI} \models t = t'$ *iff* $(\mathrm{L} \models t = t'$ *for all* $\mathrm{L} \in \mathcal{L})$*;*
- *for all predicates pr of $LT\Sigma$,*
  $\mathrm{LI} \models pr(t_1, \ldots, t_n)$ *iff* $(\mathrm{L} \models pr(t_1, \ldots, t_n)$ *for all* $\mathrm{L} \in \mathcal{L})$*; thus in particular*
  $\mathrm{LI} \models ts \xrightarrow{tl} ts'$ *iff* $(\mathrm{L} \models ts \xrightarrow{tl} ts'$ *for all* $\mathrm{L} \in \mathcal{L})$*,*
  that means that in LI *each system 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). □

Note: LT-structures introduced before may be regarded as *concrete dynamic-data types*, i.e., concrete data types of dynamic systems. Assume now that we want to abstract from the details of the concrete data, concerning the structure of (the states of) systems, of the static data and of the behaviour of the systems. By a procedure well-known in the case of static data types (see [65] also for references), we can consider *abstract dynamic-data types*, which are isomorphism classes 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 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 [7, 14]) and *dynamic specifications* (e.g., in [33, 57, 19]) for the same purpose.

Definition 5. *An* LT-specification *is 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*).* □

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 determining a class of abstract LT-structures.

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

3.2 A specification language for structured LT-specifications

In this subsection we introduce a specification language for writing LT-specifications in a structured and modular way, which is a sufficiently powerful subset of Casl, see [51].

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

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

SP_EXPR and **S** are defined inductively by the rules marked with (•) 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).

Basic specifications Basic (or flat, or unstructured) specifications are the starting building blocks, thus:

(•) $(LT\Sigma, AX) \in SP\_EXPR$ 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 following CASL syntax

sorts S
dsorts Ds_1 **label** Ds_1_Lab
...
Ds_n **label** Ds_n_Lab
ops OP
preds PR
axioms AX

to present the basic LT-specification $(LT\Sigma, AX)$, where
$LT\Sigma = (\Sigma, \{Ds_1, \ldots, Ds_n\})$ and
$\Sigma = (S \cup \{Ds_1, Ds_1\_Lab, \ldots, Ds_n, Ds_n\_Lab\}, OP,$
$PR \cup \{\ \longrightarrow : Ds_i \times Ds_i\_Lab \times Ds_i \mid i = 1, \ldots, n\})$;
i.e., the canonical transition predicates are given implicitly.

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

(•) SP$_1$ **and** SP$_2$ $\in SP\_EXPR$ for all SP$_1$, SP$_2$ $\in SP\_EXPR$;
$\mathbf{S}[\mathrm{SP}_1$ **and** $\mathrm{SP}_2] =$
$(LT\Sigma, \{\mathrm{L} \mid \mathrm{L}$ 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 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 SP_1 **and** SP_2 we obtain the wanted specification.

In practice it is often useful to use the derived construct *specification extension* defined by:

SP' **then sorts** S **dsorts** DS **ops** OP **preds** PR **axioms** AX $\quad =_{\mathrm{def}}$
let $((S', OP', PR'), DS') = Sig(\mathbf{S}[\mathrm{SP}'])$ in
SP'**and** (**sorts** $S \cup S'$ **dsorts** $DS \cup DS'$ **ops** $OP \cup OP'$ **preds** $PR \cup PR'$ **axioms** AX)

Renaming The renaming 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).

(•) SP **with** $\rho \in SP\_EXPR$
for all $\mathrm{SP} \in SP\_EXPR$ and all LT-signature isomorphisms ρ;
$\mathbf{S}[\mathrm{SP}$ **with** $\rho] =$
if ρ is an isomorphism from $Sig(\mathbf{S}[\mathrm{SP}])$ into $LT\Sigma'$ then
$(LT\Sigma', \{\mathrm{L}' \mid \mathrm{L}'$ is an $LT\Sigma'$-structure and $\rho^{-1}(\mathrm{L}') \in Mod(\mathbf{S}[\mathrm{SP}])\})$
else undefined,
where if $LT\Sigma = Sig(\mathbf{S}[\mathrm{SP}])$, 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 $LT\Sigma$.

In CASL a signature isomorphism ρ s.t. for $i = 1, \ldots, n$ $\rho(id_i) = id'_i$ and $\rho(id) = id$ for all other id is simply written
$id_1 \mapsto id'_1 \ldots id_n \mapsto id'_n$.

Reveal We use the reveal (export) construct to specify which parts of a specification (sorts, dynamic sorts, operations and predicates) should be visible from outside. Alternatively, we can specify which parts should be hidden (namely, the non-revealed ones).

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

With this construct we can act on a dynamic sort, say Ds, in two ways: we can reveal Ds completely (as in the classical setting);

or reveal its static features only (this can be obtained, when using **reveal**, 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).

Generation constraint The generation constraints allows to restrict the models of a specification to the term-generated ones (or reachable); here we consider a more general form that allows us to restrict the models to those *partially term-generated*, i.e., where only some of the operations act as constructors. In CASL the generation constraints have a particular syntactic form: to require that in a basic specification the elements of some sorts are partially generated by a set of operations, we have to group such sorts and operation declarations with curly brackets and to prefix them with the keyword **generated**.

If in a basic specification SP the sorts S and the operations C are within a generation constraint, then the semantics of SP is now

$(Sig(\mathbf{S}[\mathrm{SP}']), \{\mathrm{L} \mid \mathrm{L} \in Mod(\mathbf{S}[\mathrm{SP}'])$ and L is S-C-generated $\})$

where SP′ is SP where we have dropped the generation constrain (see Appendix A for the definition of S-C-generated).

Initial constraint The initial (free) constraint allows to restrict the models of a specification to the initial ones, if any.

(•) **free** $\{\mathrm{SP}\} \in SP\_EXPR$
for all $\mathrm{SP} \in SP\_EXPR$;
$\mathbf{S}[\textbf{free}\ \{\mathrm{SP}\}] \;=\; (LT\Sigma, \{\mathrm{L} \mid \mathrm{L} \in \text{is initial in } Mod(\mathbf{S}[\mathrm{SP}])\})$

Data types

CASL offers a compact way to declare a sort together the operations for representing its elements, the *data type declaration.*

$\textbf{type}\ Srt\ ::=\ op_1 : Srt^1_1 \times \ldots \times Srt^1_{n_1} \mid \ldots \mid op_m : Srt^m_1 \times \ldots \times Srt^m_{n_m}$

stands for

$\textbf{sort}\ Srt$
$\textbf{ops}\ op_1 : Srt^1_1 \times \ldots \times Srt^1_{n_1} \to Srt$
$\ldots$
$op_m : Srt^m_1 \times \ldots \times Srt^m_{n_m} \to Srt$

Similarly the dynamic data type declaration allows to declare a dynamic sort together the operations for representing its elements and the elements of the associated label sort.

$\textbf{dtype}\ Ds\ ::=\ op_1 : Srt^1_1 \times \ldots \times Srt^1_{n_1} \mid \ldots \mid op_m : Srt^m_1 \times \ldots \times Srt^m_{n_m}$
$\textbf{label}\ Lab\ ::=\ op'_1 : Srt'^1_1 \times \ldots \times Srt'^1_{n_1} \mid \ldots \mid op'_k : Srt'^k_1 \times \ldots \times Srt'^k_{n_k}$

stands for

dsort Ds **label** Lab
ops $op_1 : Srt^1_1 \times \ldots \times Srt^1_{n_1} \to Ds$
$\ldots$
$op_m : Srt^m_1 \times \ldots \times Srt^m_{n_m} \to Ds$
$op'_1 : Srt'^1_1 \times \ldots \times Srt'^1_{n_1} \to Lab$
$\ldots$
$op'_k : Srt'^k_1 \times \ldots \times Srt'^k_{n_k} \to Lab$

3.3 Conditional LT-specifications

At the design level we consider only conditional LT-specifications, because 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).* □

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 a sound and complete deductive system for conditional formulae).*

Proof. See [39]. □

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

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

- $\wedge_{i=1,\ldots,n}\ \alpha_i \ \Rightarrow\ \alpha$, 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*,
- $\wedge_{i=1,\ldots,n}\ \alpha_i \ \Rightarrow\ s \xrightarrow{l} s'$ is a *dynamic axiom.*

If static and dynamic axioms are separated, i.e., the transition predicates do not appear in the static axioms, then we have another characterization of the initial model.

Consider for simplicity the case of specifications with only one dynamic sort, say Ds (i.e., $DS = \{Ds\}$) and assume

$$\mathrm{S}\textsc{tat}(\mathrm{S}\textsc{p}) = (LT\Sigma, CAX - \{\text{dynamic axioms in } CAX\})$$

and $\mathrm{SI} = \mathrm{I}_{\mathrm{S}\textsc{tat}(\mathrm{S}\textsc{p})}$ (which exists); then $\mathrm{I}_{\mathrm{S}\textsc{p}}$ determines the lts: $(\mathrm{SI}_{Ds}, \mathrm{SI}_{Ds\_Lab}, \rightarrow)$, with $\rightarrow$ defined by the inductive rules

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

for all dynamic axioms $\wedge_{i=1,\ldots,n} \ s_i \xrightarrow{l_i} s'_i \ \wedge \ cond \ \Rightarrow \ s \xrightarrow{l} s' \in CAX$, 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).

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.

Example 1 (Higher-order CCS). We give a conditional LT-specification for a simple concurrent calculus, more or less the finite higher-order Milner's CCS.

spec HCCS =
free {
  INT **then**
  type $Message$ ::= $mi : Int \mid mb : Beh$
  dtype
    Beh ::= $nil \mid \_\_ . \_\_ : label \times Beh \mid \_\_ + \_\_, \ \_\_ \,||\, \_\_ : Beh \times Beh$
  label
    $label$ ::= $TAU \mid OUT, IN : Message$
  vars $b, b_1, b_2, b_3, b'_1, b'_2 : Beh; \quad l : label; \quad m : Message;$
  axioms
    $b_1 + b_2 = b_2 + b_1 \qquad (b_1 + b_2) + b_3 = b_1 + (b_2 + b_3)$
    $b + nil = b \qquad b + b = b$
    $b_1 \,||\, b_2 = b_2 \,||\, b_1 \qquad (b_1 \,||\, b_2) \,||\, b_3 = b_1 \,||\, (b_2 \,||\, b_3)$
    $b \,||\, nil = b$
  %% dynamic axioms
    $l \,.\, b \xrightarrow{l} b$
    $b_1 \xrightarrow{l} b'_1 \ \Rightarrow \ b_1 + b_2 \xrightarrow{l} b'_1$
    $b_1 \xrightarrow{l} b'_1 \ \Rightarrow \ b_1 \,||\, b_2 \xrightarrow{l} b'_1 \,||\, b_2$
    $b_1 \xrightarrow{OUT(m)} b'_1 \ \wedge \ b_2 \xrightarrow{IN(m)} b'_2 \ \Rightarrow \ b_1 \,||\, b_2 \xrightarrow{TAU} b'_1 \,||\, b'_2$
}

%% introduces a CASL comment. INT is the usual algebraic specification of integers. The first group of axioms express static properties of the behaviours; whereas the latter express dynamic properties of the behaviours by defining their transitions.

4 A Non-toy Example

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

4.1 The concurrent system CS

CS is a concurrent system made of a shared buffer and of processes cooperating among them and interacting with the external (w.r.t. the system) world by exchanging messages. The number of processes of CS is variable because the processes may terminate and new processes may be created.

The CS 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 CS (consisting of similar systems) sending and receiving messages in a broadcasting mode. The messages used by the processes are just values, and the values are integers, booleans, arrays of values and also the processes themselves. The last point allows, for example, that a process may be received by another one or from the external world, stored in the local memory and used afterwards for creating a new component of the system (thus in CS we have a limited form of mobile processes).

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

$$
\begin{array}{lll}
c & ::= & \mathit{write}(x) \mid \mathit{read}(x) \mid \mathit{is\_empty}(x) \mid \qquad (1)\\
 & & \mathit{send}(x, \mathit{ch}) \mid \mathit{rec}(x, \mathit{ch}) \mid \mathit{b\_send}(x) \mid \mathit{b\_rec}(x) \mid \qquad (2)\\
 & & \mathit{start}(x) \mid \mathit{stop} \mid \qquad (3)\\
 & & c_1 ; c_2 \mid c_1 + c_2 \mid \qquad (4)\\
 & & \mathit{skip} \mid \mathit{seq\text{-}c} \qquad (5)
\end{array}
$$

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

Thus a CS process may

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

The buffer is organized as an unbounded queue. We assume that only a testing may be performed simultaneously with a reading or a writing of the buffer, while reading and writing are mutually exclusive.

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.

NET is a distributed system consisting of several instances of CS in parallel, which interact by exchanging messages in a broadcasting mode and which perform their activity in a completely free parallel mode.

4.2 CS specification

We specify CS introduced in the previous subsection by a design LT-specification, thus with initial semantics and so corresponding to one abstract LT-structure (i.e., to one lts).

CS is a structured system and so we must first find its components (active and passive) and specify them. The active ones are the processes and the passive one is the buffer; and they are specified respectively in subsections 4.2.1 and 4.2.2. Afterwards we must specify the labels of the lts modelling CS (i.e., the interactions of CS with its external world), the states (i.e., the relevant situations in the live of CS) and the transitions of the same lts (the CS activity); such specifications are reported respectively in subsections 4.2.3, 4.2.4 and 4.2.5.

4.2.1 CS processes

The CS processes are simple systems and so we must specify the labels of the lts modelling them (i.e., the interactions of the processes with their external world), the states (i.e., the relevant situations in the live of the processes) and the transitions of the same lts (the processes activity); such specifications are reported in the following paragraphs. Furthermore, before of that we must specify the basic data handled by the processes, which are just the values.

Basic data

The values handled by the CS processes are integers, booleans and arrays of values, but also the processes themselves are values.

spec VALUE =
  INT **and** BOOL **then**
  sort *Proc*
  type $Value ::= error \mid emi : Int \mid emb : Bool \mid emp : Proc \mid$
    $\{[\_, \ldots, \_] : \underbrace{Value \times \ldots \times Value}_{n \text{ times}} \mid n \geq 2\}$
  ops $\{sel_n : Value \to Value \mid n \geq 1\}$
  vars $v_1, \ldots, v_n, \ldots, v_m : Value; \quad b : Bool; \quad i : Int; \quad p : Proc;$
  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, because when VALUE will be used in the following to build up the CS 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 union SP **and** 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 union, see Sect. 3.2).

Processes states

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

spec COMMAND =
  VID **and** CHID **then**
  type *Command* ::=
    $write, read, is\_empty, b\_send, b\_rec, start : Vid \mid$
    $send, rec : Vid \times Chid \mid skip \mid stop \mid$
    $\_;\_,\ \_ + \_ : Command \times Command \mid seq_1 : \ldots \mid \ldots \mid seq_n : \ldots$
  vars $c, c_1, c_2, c_3 : Command$;
  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, because they are not relevant. $seq_1, \ldots, seq_n$ are the operations defining the sequential commands.

spec LMEM =
  VID **and** VALUE **then**
  type *Lmem* ::= $[] \mid \_[\_/\_] : Lmem \times Value \times Vid$
  op $\_(\_) : Lmem \times Vid \rightarrow Value$
  vars $x, x' : Vid; \quad lm : Lmem; \quad v : Value;$
  axioms
    $[](x) = error$
    $lm[v/x](x) = v$
    $x\ Dif\ x' \Rightarrow lm[v/x](x') = lm(x')$

spec PROC_STATE =
  COMMAND **and** LMEM **then**
  type *Proc* ::= $\langle \_, \_ \rangle : Command \times Lmem$

Processes labels

The labels of the lts modelling the CS processes are in correspondence with the executions of the concurrent commands except TAU, which corresponds to the execution of the sequential commands.

spec PROC_LAB =
  PROC_STATE **then**
  type *Proc_Lab* ::= $WRITE, READ, IS\_EMPTY, B\_SEND, B\_REC : Value \mid$
    $SEND, REC : Value \times Chid \mid START : Proc \mid TAU \mid STOP$

Processes activity

Finally, we give the activity of the CS processes, by axiomatically defining the transitions of the lts modelling them.

spec PROC =
  PROC_STATE **and** PROC_LAB **then**
  dsort *Proc* **label** *Proc_Lab*

vars $x : Vid$; $v : Value$; $lm, lm' : Lmem$; $ch : Chid$;
$p : Proc$; $lp : Proc\_Lab$; $c_1, c_2, c'_1 : Command$;
axioms
%% 1

$$\langle write(x), lm\rangle \xrightarrow{WRITE(lm(x))} \langle skip, lm\rangle$$
$$\langle read(x), lm\rangle \xrightarrow{READ(v)} \langle skip, lm[v/x]\rangle$$
$$\langle is\_empty(x), lm\rangle \xrightarrow{IS\_EMPTY(v)} \langle skip, lm[v/x]\rangle$$

%% 2

$$\langle send(x, ch), lm\rangle \xrightarrow{SEND(lm(x), ch)} \langle skip, lm\rangle$$
$$\langle rec(x, ch), lm\rangle \xrightarrow{REC(v, ch)} \langle skip, lm[v/x]\rangle$$

%% 3

$$\langle b\_send(x), lm\rangle \xrightarrow{B\_SEND(lm(x))} \langle skip, lm\rangle$$
$$\langle b\_rec(x), lm\rangle \xrightarrow{B\_REC(v)} \langle skip, lm[v/x]\rangle$$

%% 4

$$lm(x) = emp(p) \;\Rightarrow\; \langle start(x), lm\rangle \xrightarrow{START(p)} \langle skip, lm\rangle$$
$$\langle stop, lm\rangle \xrightarrow{STOP} \langle skip, []\rangle$$

%% 5

$$\langle c_1, lm\rangle \xrightarrow{lp} \langle c'_1, lm'\rangle \;\Rightarrow\; \langle c_1; c_2, lm\rangle \xrightarrow{lp} \langle c'_1; c_2, lm'\rangle$$
$$\langle c_1, lm\rangle \xrightarrow{lp} \langle c'_1, lm'\rangle \;\Rightarrow\; \langle c_1 + c_2, lm\rangle \xrightarrow{lp} \langle c'_1, lm'\rangle$$

%% 6

$$\ldots \;\Rightarrow\; \langle seq_1(\ldots), lm\rangle \xrightarrow{TAU} \langle \ldots, \ldots\rangle$$
$$\ldots$$

Above "__(__)" and "__[__/__]" denote respectively the selection of the associate element and the updating operation of the specification LMEM.

The axiom group 1 defines the writing, reading and testing of the emptiness of the buffer; groups 2 ad 3 define respectively the handshaking communications and the broadcasting communications outside CS. The axiom group 4 define the creation of a new process defined by the value contained in the local variable x (if is a process) and the immediate termination of a process. The axiom group 5 defines the sequential composition and the nondeterministic choice; notice that these two axioms are sufficient because ";" is associative, "$skip; c = c$", and "+" is commutative (from the static axioms in COMMAND). Several axioms having the form 6, not further detailed here, define the sequential commands.

4.2.2 Buffer

The buffer is the passive component of CS, and in this case is organized as a queue. Because we assume that the activity of a passive component just consists of unconditionally passing from a state to whatever other state, we simply specify its states, avoiding to explicitly describe the transitions. However to guarantee an appropriate abstraction level the states will be specified by an abstract data type.

spec BUFFER =
  VALUE **then**
  type $Buffer$ ::= $empty \mid put : Value \times Buffer$
  ops $first : Buffer \rightarrow Value$
    $get : Buffer \rightarrow Buffer$
  pred $not\_empty : Buffer$
  vars $v : Value; \quad bf : Buffer;$
  axioms
    $first(empty) = error \qquad first(put(v, empty)) = v$
    $not\_empty(bf) \Rightarrow first(put(v, bf)) = first(bf)$
    $get(empty) = empty \qquad get(put(v, empty)) = empty$
    $not\_empty(bf) \Rightarrow get(put(v, bf)) = put(v, get(bf))$
    $not\_empty(put(v, bf))$

The operation *empty* denotes the empty queue, *put* is the operation for adding a value to the queue and *first*, *get* those for getting and dropping the first element from a queue.

4.2.3 CS labels

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

spec CS_LAB =
  VALUE **then**
  type Cs_Lab ::= $TAU \mid BS, BR : Value \mid \_\&\_ : Cs\_Lab \times Cs\_Lab$
  vars $l_1, l_2, l_3 : Cs\_Lab;$
  axioms
    $l_1 \,\&\, l_2 = l_2 \,\&\, l_1 \qquad TAU \,\&\, l_1 = l_1$
    $(l_1 \,\&\, l_2) \,\&\, l_3 = l_1 \,\&\, (l_2 \,\&\, l_3)$

4.2.4 CS states

The states of the active and passive components of CS (i.e., the processes and a buffer) determine the states of the lts modelling CS itself; because the order of the processes is not relevant and it is possible to have more than one process in the same state, the CS states are pairs whose components are a multiset of process states and a buffer state.

spec CS_STATE =
  BUFFER **and** MSET[PROC] **then**
  type Cs ::= $\_ \mid \_ : Mset[Proc] \times Buffer$

Here and in the following we use the parametric specification MSET[_] reported below.

spec ELEM =
  sort *Elem*

spec MSET[ELEM] =
  type $Mset[Elem]$ ::= $\{\}$ | $\{\_\_\} : Elem$ | $\_\_ \mid \_\_ : Mset[Elem] \times Mset[Elem]$
  vars $ms, ms_1, ms_2, ms_3 : Mset[Elem]$;
  axioms
    $ms \mid \{\} = ms$
    $ms_1 \mid ms_2 = ms_2 \mid ms_1$
    $(ms_1 \mid ms_2) \mid ms_3 = ms_1 \mid (ms_2 \mid ms_3)$

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

4.2.5 CS activity

The transitions of the lts modelling CS (i.e., the CS 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 that MUST be performed together) that in turn MAY be performed together; then these partial moves are used to give the CS transitions. The partial moves are still formalized as labelled transitions (defined by an auxiliary predicate $\_\_ \overline{\sim\sim\sim>} \_\_$) 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_access* is a predicate checking whether a partial move does not involve an exclusive buffer access (either reading or writing it).

spec AUX =
  VALUE **then**
  type Aux ::= $TAU \mid B\_ACCESS \mid B\_SEND, B\_REC : Value \mid$
    $\_\_ \,//\, \_\_ : Aux \times Aux$
  pred $no\_access : Aux$
  vars $l_1, l_2, l_3 : Aux$; $v : Value$;
  axioms
    $l_1 \,//\, l_2 = l_2 \,//\, l_1$ $(l_1 \,//\, l_2) \,//\, l_3 = l_1 \,//\, (l_2 \,//\, l_3)$
    $TAU \,//\, l_1 = l_1$ $B\_ACCESS \,//\, B\_ACCESS = B\_ACCESS$
    $no\_access(TAU)$ $no\_access(B\_SEND(v))$ $no\_access(B\_REC(v))$
    $no\_access(l_1) \wedge no\_access(l_2) \Rightarrow no\_access(l_1 \,//\, l_2)$

spec CS_SPEC =
  CS_STATE **and** CS_LAB **and** AUX **then**
free {
  dsort *Cs* **label** *Cs_Lab*
  op $out : Aux \rightarrow Cs\_Lab$
  pred $\_\_ \overline{\sim\sim\sim>} \_\_ : Cs \times Aux \times Cs$
  vars $p, p', p_1, p_1', p_2, p_2' : Proc$; $l_1, l_2 : Proc\_Lab$; $v : Value$;
    $bf, bf' : Buffer$; $ch : Chid$;
    $pms_1, pms_2, pms_1', pms_2', pms, pms' : Mset[Proc]$;

axioms
%% 1
$$p \xrightarrow{TAU} p' \;\Rightarrow\; p \mid bf \overset{TAU}{\rightsquigarrow} p' \mid bf$$
%% 2
$$p \xrightarrow{READ(v)} p' \;\wedge\; first(bf) = v \;\Rightarrow\; p \mid bf \overset{B\_ACCESS}{\rightsquigarrow} p' \mid get(bf)$$
$$p \xrightarrow{WRITE(v)} p' \;\Rightarrow\; p \mid bf \overset{B\_ACCESS}{\rightsquigarrow} p' \mid put(v, bf)$$
$$p \xrightarrow{IS\_EMPTY(False)} p' \;\wedge\; not\_empty(bf) \;\Rightarrow\; p \mid bf \overset{TAU}{\rightsquigarrow} p' \mid bf$$
$$p \xrightarrow{IS\_EMPTY(True)} p' \;\Rightarrow\; p \mid empty \overset{TAU}{\rightsquigarrow} p' \mid empty$$
%% 3
$$p_1 \xrightarrow{REC(v,ch)} p_1' \;\wedge\; p_2 \xrightarrow{SEND(v,ch)} p_2' \;\Rightarrow$$
$$p_1 \mid p_2 \mid bf \overset{TAU}{\rightsquigarrow} p_1' \mid p_2' \mid bf$$
%% 4
$$p_1 \xrightarrow{START(p)} p_1' \;\Rightarrow\; p_1 \mid bf \overset{TAU}{\rightsquigarrow} p_1' \mid p \mid bf$$
$$p \xrightarrow{STOP} p_1' \;\Rightarrow\; p \mid bf \overset{TAU}{\rightsquigarrow} \{\} \mid bf$$
%% 5
$$p \xrightarrow{B\_SEND(v)} p' \;\Rightarrow\; p \mid bf \overset{B\_SEND(v)}{\rightsquigarrow} p' \mid bf$$
$$p \xrightarrow{B\_REC(v)} p' \;\Rightarrow\; p \mid bf \overset{B\_REC(v)}{\rightsquigarrow} p' \mid bf$$
%% 6
$$pms_1 \mid bf \overset{l_1}{\rightsquigarrow} pms_1' \mid bf \;\wedge\; no\_access(l_1) \;\wedge$$
$$pms_2 \mid bf \overset{l_2}{\rightsquigarrow} pms_2' \mid bf' \;\Rightarrow$$
$$pms_1 \mid pms_2 \mid bf \overset{l_1\;//\;l_2}{\rightsquigarrow} pms_1' \mid pms_2' \mid bf'$$
%% 7
$$pms \mid bf \overset{l}{\rightsquigarrow} pms' \mid bf' \;\Rightarrow$$
$$pms \mid pms_1 \mid bf \xrightarrow{out(l)} pms' \mid pms_1 \mid bf'$$
%% 8
$$out(TAU) = TAU \qquad out(B\_ACCESS) = TAU$$
$$out(B\_SEND(v)) = BS(v) \qquad out(B\_REC(v)) = BR(v)$$
$$out(l_1\;//\;l_2) = out(l_1)\;\&\;out(l_2)$$
}

The axiom 1 states that each process internal action capability becomes a partial move. The axioms 2 state that an action capability of form $READ(v)$ becomes a partial move only if v is the first available element of the buffer, that an action capability of form $WRITE(v)$ always becomes a partial move putting the value v on the buffer, and 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 that sends the message and the one that receives it (axiom 3).

The axioms 4 defines the creation of a new process and the termination of an exiting one.

The axioms 5 state that the action capabilities corresponding to broadcasting communications, can always become partial moves.

The axiom 6 states that each partial move

$$pms_1 \mid bf \overset{l_1}{\rightsquigarrow\!\!\rightsquigarrow\!\!>} pms'_1 \mid bf$$

not involving an exclusive buffer access (either reading or writing it) may be performed together with whatever other group of partial moves; as a consequence we have that an exclusive buffer access cannot be performed together with another similar access.

The axiom 7 states that each group of partial moves may become a transition of CS; notice that here pms_1 is the set of the process components that do not take part in the transition (recall that CS 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 8).

4.3 CS variants

In this subsection, to show the modular features of LTL/SMoLCS we give some hints on how changes of the assumptions on CS reported in Sect. 4.1 may be reflected in the specification CS_SPEC by changing some of its parts.

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 CS it is sufficient to drop the sort *Proc* and the operation *emp* from the original specification VALUE.

Buffer If the buffer component of the system 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 first six axioms of BUFFER with:

$$first(empty) = error \qquad first(put(v, bf)) = v$$
$$get(put(v, bf)) = bf \qquad get(empty) = empty;$$

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

$put(v, put(v', bf)) = put(v, bf)$
$first(put(v, empty)) = v \qquad first(empty) = error$
$get(put(v, empty)) = empty \qquad get(empty) = empty$

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 $\_ \overset{-}{\leadsto} \_$.

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

$$p \xrightarrow{IS\_EMPTY(False)} p' \wedge not\_empty(bf) \Rightarrow p \mid bf \overset{B\_ACCESS}{\leadsto} p' \mid bf$$
$$p \xrightarrow{IS\_EMPTY(True)} p' \Rightarrow p \mid empty \overset{TAU}{\leadsto} p' \mid empty$$

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 6. For example, if the CS becomes an interleaving system, then it is sufficient to drop such axiom.

4.4 Specification of NET

We formally define NET with the design LT-specification NET_Spec given below.

NET is a structured system, whose active components are (variants of) the CS system described by the LT-specification CS_Spec given either in subsection 4.2 or in 4.3, and without passive components. Thus the states of the lts modelling NET are just multisets of states of CS systems. Because NET is a closed system (no interactions with the external world) its transitions are all labelled by TAU. In NET only one broadcasting communication may be performed at each time, and there are no restrictions on the internal activities of the component systems, so its activity is simply defined below.

spec NET_Spec =
free {
  Mset[CS_Spec] **with** *Mset[Cs]* $\mapsto$ *Net* **then**
  dsort *Net* **label** *Net_Lab*
  type *Net_Lab* ::= *TAU*
  vars $c, c', c_1, c_1', \ldots, c_n, c_n', c_{n+1}, c_{n+1}', \ldots, c_m, c_m' : Cs; \quad v : Value;$
  axioms
  %% 1
  $\{c \xrightarrow{B\_SEND(v)} c' \wedge c_1 \xrightarrow{B\_REC(v)} c_1' \wedge \ldots \wedge c_n \xrightarrow{B\_REC(v)} c_n' \wedge$
  $c_{n+1} \xrightarrow{TAU} c_{n+1}' \wedge \ldots \wedge c_m \xrightarrow{TAU} c_m' \Rightarrow$

$$c \mid c_1 \mid \ldots \mid c_n \mid c_{n+1} \mid \ldots \mid c_m \mid cms \xrightarrow{TAU}$$
$$c' \mid c'_1 \mid \ldots \mid c'_n \mid c'_{n+1} \mid \ldots \mid c'_m \mid cms$$
$$\mid \; n, m \geq 0\}$$

%% 2

$$\{c_1 \xrightarrow{TAU} c'_1 \;\wedge\; \ldots \;\wedge\; c_n \xrightarrow{TAU} c'_n \;\Rightarrow\; c_1 \mid \ldots \mid c_n \mid cms \xrightarrow{TAU}$$
$$c'_1 \mid \ldots \mid c'_n \mid cms$$
$$\mid \; n \geq 1\}$$

}

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

Notice that a CS system may also perform several broadcasting communications simultaneously; but these capabilities will never become effective when it is a component of 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 requirements about systems formally modelled by LT-structures.

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

The common and relevant requirements on a system can be distinguished as follows.

Static requirements, i.e., about the structure of the systems, of their active and 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 components there are and how they are put together to build a structured system) 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 systems. They are expressed by the static axioms about the labels, 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 (i.e., concerning labelled transitions, formally those in which the transition predicates appear), for example, a system in a certain state has just two capabilities of moving to two other states or it cannot have some capabilities.

Example 2 (Requirements on concurrent systems). Here we show how to formally express some requirements on concurrent systems similar to CS, 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 =
  dsort *Proc* **label** *Proc_Lab*
  type *Value* ::= *value* : *Proc*
  free type *Proc_Lab* ::= *WRITE*, *READ*, *OUT*, *IN* : *Value* | *TAU*
  vars $p, p_1, p', p'_1 : Proc;\quad lp : Proc\_Lab;$
  axioms

$$p \xrightarrow{OUT(value(p_1))} p' \;\vee\; p \xrightarrow{IN(value(p_1))} p' \;\Rightarrow\; \exists\, lp, p'_1 \bullet p_1 \xrightarrow{lp} p'_1$$

The CASL construct **free type** is a shortcut corresponding to declare a data type and to require that all the operations of the type are injective, that the elements built by two different operations are always different and that all elements are generated by some of the listed operations.

The above specification requires that the processes can perform five different kinds of actions (described by the operations of sort *Proc_Lab*) using values that may be the same processes (by means of the operation *value*). Notice that however the values are not fully specified. The last axiom states that only non-stopped processes can be communicated.

spec CS_REQ =
  MSET[PROC_REQ] **then**
  dsort *Cs* **label** *Cs_Lab*
  sort *Buffer*
  preds *empty*, *broken* : *Buffer*
  free type Cs ::= $\_\_ \mid \_\_ : Mset[Proc] \times Buffer$
  vars $c : Cs;\quad lc : Cs\_Lab;\quad p, p' : Proc;\quad lp : Proc\_Lab;$
    $pms, pms' : Mset[Proc];\quad bf : Buffer;$
  axioms
  %% 1

$$(\neg\, \exists\, lc, c \bullet pms \mid bf \xrightarrow{lc} c) \;\Rightarrow$$
$$(\neg\, \exists\, pms', p, p', lp \bullet pms = p \mid pms' \;\wedge\; p \xrightarrow{lp} p')$$

  %% 2

$(\neg\ \exists\ c, lc \bullet c \xrightarrow{lc} pms \mid bf) \ \Rightarrow\ \ empty(bf)$

%% 3

$\neg\ \exists\ c, lc, bf \bullet (pms \mid bf \xrightarrow{lc} c \ \wedge\ broken(bf))$

The free type construct used to define the sort *Cs* formalizes a requirement on the structure of the concurrent systems; 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 Sect. 4.2.4) formalize that the order of the process components is not relevant. No deadlocks are allowed in the specified concurrent systems (axiom 1). Initially the buffer must be empty (axiom 2). The broken buffer stops the whole system (axiom 3).

From the axioms of CS_REQ we can prove (using usual first-order stuff) that the axioms 1 and 3 imply

$\forall\ p \bullet \neg\exists\ p', lp \bullet p \xrightarrow{lp} p'$,

i.e., that all processes are terminated, which makes the above specification not very interesting. So we can replace axiom 1 with

$$(1') \qquad ((\neg\ broken(bf)\ \wedge\ (\neg\ \exists\ lc, c \bullet pms \mid bf \xrightarrow{lc} c)) \ \Rightarrow \\ (\neg\ \exists\ pms', p, p', lp \bullet pms = p \mid pms' \ \wedge\ p \xrightarrow{lp} p')$$

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

5.2 More expressive logic

First-order logic allows to express only few requirements on systems, similar to those given in Ex. 2, 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 a system is not relevant) and limited properties on the activity of the systems (e.g., a terminated process cannot be communicated). However, such logic cannot express a great number of more interesting and relevant properties, as the following one.

1. only the processes that in any case will terminate may be communicated;
2. during each possible execution each system eventually will reach a situation where the buffer is empty;
3. the unbreakable buffer does not exist (i.e., there does not exist a buffer that in any case will never become broken);
4. no deadlocks are allowed but without fixing the structure of the concurrent systems (i.e., we want to give a specification with admits, as models, for example, both systems 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 and liveness properties and abstract properties on the concurrent structure).

For what concerns properties on the behaviour, [33] presents an integration of the combinators of branching-time temporal logic in LTL for expressing the safety and liveness properties; [57, 17] propose a new logic based on the concept of abstract event for expressing overall properties on the activity of the systems (e.g., a certain activity, not necessarily atomic [i.e., consisting of one state or one transition], will be eventually followed by the sequential composition of two other non-atomic activities), and [32] shows an integration with the permission and obligation combinators of the deontic logic.

Instead, in [56, 16] we have proposed entity specifications for handling properties on the system structure. In an entity specification we can express properties on the concurrent/distributed structure of a system by using a special predicate *are_sub* s.t. *dset are_sub d* is true whenever *dset* (a set of systems) are all the active components of d (a system).

Using the LTL variants proposed in the papers quoted above, we may formalize the properties 1, ..., 4 as follows, where $\Diamond$, $\Box$ are the usual temporal combinators "eventually" and "always" including the present and *in_any_case* is the CTL-style temporal combinator "for all paths". Notice that, because we consider different systems also of different types, the formulae built by *in_any_case* are anchored to a term of dynamic sort (representing the system to whom the temporal property is referring).

1 $(p \xrightarrow{OUT(value(p_1))} p' \;\vee\; p \xrightarrow{IN(value(p_1))} p') \;\Rightarrow$
$in\_any\_case(p_1, \Diamond[x \bullet \neg \exists\, p'', lp \bullet x \xrightarrow{lp} p''])$
(in any case process p_1 will reach a final state, i.e., where it cannot move anymore).

2 $in\_any\_case(c, \Diamond[x \bullet \exists\, pms, bf \bullet x = pms \mid bf \;\wedge\; empty(bf)])$

3 $\neg \exists\, bf \bullet in\_any\_case(bf, \Box[bf' \bullet \neg\, broken(bf')])$

4 $(\neg \exists\, c', lc \bullet c \xrightarrow{lc} c') \;\wedge\; dset\ are\_sub\ c \;\wedge\; d \in dset \;\Rightarrow$
$\neg \exists\, d', l' \bullet d \xrightarrow{l} d'$

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 [65]); we have chosen this notion because 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); whereas implementation as refinement is obtained by requiring the 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 extension with derived operations and predicates, a reveal and an extension with axioms, we have the so called implementation by rename-extend-restrict-identify of [35, 36]; which corresponds, within the framework of abstract data types, to the 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* $\mathrm{SP} = (LT\Sigma, AX)$ *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* $\mathrm{SP} \rightsquigarrow^{f} \mathrm{SP}_1$*) 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 $\textsc{Sp}'$; we have a static implementation, which concerns just the static parts of the specification (for example, either the data handled by the systems or the states and the labels of the lts modelling them);
– f redefines the transitions of $\textsc{Sp}'$ by composing them sequentially, i.e., by adding axioms like

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

($\longrightarrow$ transition predicate in $\textsc{Sp}'$ and $\Longrightarrow$ transition predicate in $\textsc{Sp}$); we have an action refining implementation, because the transitions of the systems of $\textsc{Sp}$ are realized by sequences of transitions in $\textsc{Sp}'$.
– f does not change dynamic sorts into static ones; we have a dynamics-preserving implementation.

Example 3 (Implementations of CS_REQ′*).* We consider some implementations of the requirement LT-specification CS_REQ′ (with the axiom 1′) introduced in Ex. 2, because there is no room to introduce completely new and more interesting examples.

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

- by saying how the various parts of CS_REQ are realized using those of CS_SPEC, precisely
  – by extending CS_SPEC with the predicates and operations
    broken, *empty*: *Buffer*,
    OUT, *IN*: *Value* → *Proc_Lab*
    defined by the axioms
    broken(*bf*), for all *bf* including an *error* value,
    empty(*bf*), for all *bf* = *empty*,
    $SEND(v) = OUT(v), B\_SEND(v) = OUT(v)$,
    $REC(v) = IN(v), B\_REC(v) = IN(v)$
  – by renaming *value* as *emp*;
- by hiding all operations and sorts present in CS_SPEC and not in CS_REQ′ (so the signatures of f(CS_SPEC) and CS_REQ are the same).

Then, we have to check if the axioms of CS_REQ hold in f(CS_SPEC). Unfortunately, the axiom 1′ (requiring the absence of deadlocks when the buffer is not broken) does not hold, because all states of CS having only one process component with command part $send(x, ch)$ are deadlock situations.

Not even PROC is a correct implementation of PROC_REQ, because 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 CS_REQ′. In such specification the elements of various sorts (static and dynamic) as *Value*, *Proc* and *Cs* are completely defined; moreover also the activity of the systems is fully defined. As usual, first we give the specification of the components, BEHAVIOUR, and after of the whole system, SYSTEM.

spec BEHAVIOUR =
free {
  types
    $Chan ::= alpha, beta, \ldots$
    $Value ::= emb : Beh \mid emc : Chan$
  dtype
    $Beh ::= nil \mid \_?\_, \_!\_ : Chan \times Beh \mid \_ + \_ : Beh \times Beh \mid \delta\ \_ : Beh$
  label
    $label ::= OUT, IN : Value \mid TAU$
  vars $ch : Chan;\quad b, b', b_1, b_1' : Beh;\quad l : label;$
  axioms
    $+$ comm., assoc., id: nil
    $ch!b \xrightarrow{OUT(emc(ch))} b$
    $ch?b \xrightarrow{IN(emc(ch))} b$
    $b_1 \xrightarrow{l} b_1' \Rightarrow b_1 + b_2 \xrightarrow{l} b_1'$
    $\delta\ b \xrightarrow{TAU} \delta\ b$
    $b \xrightarrow{l} b' \Rightarrow \delta\ b \xrightarrow{l} b'$
}

BEHAVIOUR 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* $\rightarrow$ *Proc_Lab*, and to reveal only the operations needed in PROC_REQ. Then all properties defined by the PROC_REQ axioms are satisfied, because BEHAVIOUR behaviours never communicate other behaviours outside (no transition labelled by either $OUT(emb(b))$ or $IN(emb(b))$ will ever be performed).

spec SYSTEM =
free {
  MSET[BEHAVIOUR] *Mset[Beh]* $\mapsto$ *Mset[Proc]* **with then**
  type $Status ::= working \mid broken$
  dtype
    $System ::= \_ \mid \_ : Mset[Proc] \times Status$
  label

$$
\begin{array}{l}
\quad Beh ::= TAU \mid IN, OUT : Value \\
\textbf{vars}\ \ ch : Chan\ b, b', b_1, b_1', b_2, b_2' : Beh; \quad mb, mb_1, mb_1' : Mset[Beh]; \\
\textbf{axioms} \\
\quad b \xrightarrow{TAU} b' \ \Rightarrow\ b \mid working \xrightarrow{TAU} b' \mid working \\
\quad b_1 \xrightarrow{OUT(ch)} b_1' \ \wedge\ b_2 \xrightarrow{IN(ch)} b_2' \ \Rightarrow \\
\qquad b_1 \mid b_2 \mid working \xrightarrow{TAU} b_1' \mid b_2' \mid working \\
\quad b_1 \xrightarrow{OUT(ch)} b_1' \ \Rightarrow\ b_1 \mid working \xrightarrow{OUT(ch)} b_1' \mid working \\
\quad b_1 \xrightarrow{IN(ch)} b_1' \ \Rightarrow\ b_1 \mid working \xrightarrow{IN(ch)} b_1' \mid working \\
\quad mb_1 \mid working \xrightarrow{ls} mb_1' \mid working \ \Rightarrow \\
\qquad mb_1 \mid mb \mid working \xrightarrow{ls} mb_1' \mid mb \mid working \\
\quad mb \mid working \xrightarrow{TAU} mb \mid broken
\end{array}
$$

}

SYSTEM is an implementation of CS_REQ; indeed after making the appropriate renamings (including *working* into *empty*), extension and revealings we get a new specification where all axioms of CS_REQ are satisfied, thus whose models are included into the models of CS_REQ. The axiom $1'$ holds because no behaviour activity may be blocked when the status is *working*; the axiom 2 holds because there are no initial states and the axiom 3 holds because all axioms of SYSTEM defining transitions require the status to be equal to *working*.

The above example is a case where the implementing systems 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 following specification shows.

spec CS_REQ2 =
 CS_REQ **then**
 dsort *Agent* **label** *Agent_Lab*
 ops $\_ : Agent \to Proc$
 $\_ \mid\mid\mid \_ : Agent \times Proc \to Proc$
 axioms
 ...

In this case CS_REQ $\sim\!\sim\!\sim\!>^{f}$ CS_REQ*2*, where f is just the hiding of the dynamic sort *Agent* and of the operation $|||$; and we have that the elements of sort *Proc* (simple systems) are implemented by groups of other systems cooperating among them (i.e., structured systems).

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 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 [54, 55, 47] on SOS and CCS. Indeed, our method can be seen as a generalization of some 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 [55] a binary parallel operator gives the structure of a CSP program, although the true concurrent structure is that of an unordered group of processes (i.e., a set); and [44] 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, because 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 [7] for a first appearance and [46] 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 processes simulate 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 [1]); 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 [23]).

Most of the known methods aim at providing algebraic specifications of the static structures used by processes; among them LOTOS

[42] 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 [45], 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 [46]; see [20] 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, which are 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 is 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 specially 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 the authors claim that RL is an universal formalism for concurrency, because any other one can be subsumed by RL, but it is better to say that any other one can be coded in it in some way. This is true also for LTL; indeed in [20] 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 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., [25, 26], and projects M. Broy has developed since 1983 an approach to the formal specification of concurrent systems that 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 [43], 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 logical formalism cannot be exploited any more. Moreover, another major distinction is typing that, 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 [30] 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 ASM (Abstract State Machines) of Gurevich [40]; notice however that ASM 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 [22]; but specification issues have not yet been tackled. The issue of specification has been addressed by Dauchy and Gaudel [34]; moreover a kind of manifesto supporting the approach and building up especially on previous work in [34] has been issued by Ehrig and Orejas in [37]. 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 CS 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 CS'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 CS.

In TLA no notion of process (component of the system) is provided, and so a CS 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 CS (also if the command for creating new processes is dropped); instead we have to give *one* TLA specification for the CS with n process, for each $n \geq 1$.

Because in TLA there is no notion of type, the various data used in CS (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 CS 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 CS we have different kinds of cooperations among the CS 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 because 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 and LOTOS No way to handle the higher-order concurrent features (unless we upgrade to higher-order CCS and the like).

In general each CS 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 CS 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 CS 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 [20] we show how to code a subclass of LTL specifications into RL; our CS specification after changing some details (e.g., transforming the auxiliary transition relation $\rightsquigarrow$ 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 CS. Clearly such specification has been obtained by using the ideas, techniques and methods of LTL/SMoLCS without any reference to RL; but now, for example, we can use some prototyping tools of RL on it.

Following the ideas, techniques and methods of RL: We can follow 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 CS 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 CS example has been chosen to illustrate the features of the LTL/SMoLCS approach and, because 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 that 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 systems, 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 [47]; 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 [23]) 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, whereas 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 [14, 12]) with an important contribution of M. Wirsing in the early stages (see [7]); later on other people contributed to some related aspects, notably A. Giovini († 93) for the observational semantics [5] and F. Morando for the development of a rapid prototyping system [38, 9]. The idea of conditional LTL was the core of [7], where its specifications were called algebraic transition systems; the concept of higher-order concurrency was explored in [14] and its semantics foundations laid down in [3] and generalized in [5]. Requirement LT-specifications have been dealt with especially in [33, 57, 17] and some methodological aspects have been addressed in [18].

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 [8]. 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., [13, 2, 10]). 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 [18]. 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 and 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, although for some other applications partial or 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 [29, 28] 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 systems.

Acknowledgements. 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

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 $op \colon s_1 \times \ldots \times s_n \to s$ for $op \in OP_{s_1 \ldots s_n, s}$, $pr \colon s_1 \times \ldots \times s_n$ 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*; precisely*

- *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* $t^{\mathrm{A},V}$, *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* S *and* C *are respectively a set of sorts and of operation symbols of* Σ, *then* A *is* S-C-generated *iff for all* $s \in S$, *for all* $a \in \mathrm{A}_s$, *there exists* X *a family of variables s.t.* $X_{s'} = \emptyset$ *for all* $s' \in S$, *a variable evaluation* $V: X \to \mathrm{A}$ *and* $t \in T_{\Sigma(C)}(X)_s$ *s.t.* $t^{V,\mathrm{A}} = a$; *where:*
  - $Sorts(C) =$
    $\{s \mid$ *there exists* $op \in C$ *having* s *either as argument or as result sort* $\}$,

– $\Sigma(C)$ *is the signature* $(Sorts(C) \cup S, C, \emptyset)$ $(\subseteq \Sigma)$.

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* $h{:}\, \mathrm{A} \to \mathrm{B}$*) is a family of* total *functions* $h = \{h_s\}_{s \in S}$ *where for all* $s \in S$ $h_s{:}\, \mathrm{A}_s \to \mathrm{B}_s$ *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 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 A $\models \theta$) whenever 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 $\theta \in AX$ 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* $\phi: \mathrm{A} \to \mathrm{A}'$. □

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* $CAX \vdash pr(t_1, \ldots, t_n)$. □

References

1. E. Astesiano, M. Broy, and G. Reggio. Algebraic Specification of Concurrent Systems. In E. Astesiano, B. Krieg-Bruckner, and H.-J. Kreowski, editors, *IFIP WG 1.3 Book on Algebraic Foundations of System Specification*, pages 467 – 520. Springer Verlag, 1999.
2. 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.
3. 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.
4. 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.
5. E. Astesiano, A. Giovini, and G. Reggio. Observational Structures and their Logic. *T.C.S.*, 96:249–283, 1992.
6. 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.
7. 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.

8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
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. 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.
18. 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.
19. E. Astesiano and G. Reggio. A Dynamic Specification of the RPC-Memory Problem. In M. Broy, S. Merz, and K. Spies, editors, *Formal System Specification: The RPC-Memory Specification Case Study*, number 1169 in Lecture Notes in Computer Science, pages 67–108. Springer Verlag, Berlin, 1996.
20. E. Astesiano and G. Reggio. On the Relationship between Labelled Transition Logic and Rewriting Logic. Technical Report DISI–TR–97–23, DISI – Università di Genova, Italy, 1997.

21. E. Astesiano and G. Reggio. Formalism and Method. *T.C.S.*, (236):3–34, 2000.
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. J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, Cambridge, 1990.
24. 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.
25. 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.
26. M. Broy. Predicative Specifications for Functional Programs Describing Communicating Networks. *Information Processing Letters*, 25:2, 1987.
27. M. Broy and M. Wirsing. Partial Abstract Types. *Acta Informatica*, 18:47–64, 1982.
28. 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.
29. M. Cerioli and G. Reggio. Very abstract specifications: a formalism independent approach. *Mathematical Structures in Computer Science*, 1(8), 1998.
30. M. Chandy and J. Misra. *Parallel Program Design: a Foundation.* Addison-Wesley, 1988.
31. 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.
32. 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.
33. G. Costa and G. Reggio. Specification of Abstract Dynamic Data Types: A Temporal Logic Approach. *T.C.S.*, 173(2):513–554, 1997.
34. 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.
35. 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.
36. H. Ehrig, H.J. Kreowski, B. Mahr, and P. Padawitz. Algebraic Implementation of Abstract Data Types. *T.C.S.*, 20:209–263, 1982.
37. H. Ehrig and F. Orejas. Dynamic Abstract Data Types: An Informal Proposal. *Bulletin of the EATCS*, (53), 1994.
38. 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.

39. J.A. 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.
40. Y. Gurevich. Evolving Algebras: a Tutorial Introduction. *Bulletin of the EATCS*, (43):264–284, 1991.
41. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, London, 1985.
42. I.S.O. ISO 8807 Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS, International Organization for Standardization, 1989.
43. L. Lamport. The Temporal Logic of Actions. Technical Report 79, Digital, Systems Research Center, Palo Alto, California, 1991.
44. Wei Li. An Operational Semantics of Multitasking and Exception Handling in Ada. In *Proc. AC Ada Tech. and Tutorial Conference.* ACM Press, 1982.
45. 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.
46. J. Meseguer. Conditional Rewriting as a Unified Model of Concurrency. *T.C.S.*, 96:73–155, 1992.
47. R. Milner. *A Calculus of Communicating Systems.* Number 92 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1980.
48. R. Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.
49. 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.
50. 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.
51. P.D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT '97*, number 1214 in Lecture Notes in Computer Science, pages 115–137. Springer Verlag, Berlin, 1997.
52. R. De Nicola and M. Hennessy. Testing Equivalence for Processes. *T.C.S.*, 34:181–205, 1984.
53. 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.
54. G. Plotkin. A Structural Approach to Operational Semantics. Lecture notes, Aarhus University, 1981.
55. 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.
56. 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.
57. 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.

58. G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL : A CASL Extension for Dynamic Reactive Systems – Summary. Technical Report DISI-TR-99-34, DISI – Università di Genova, Italy, 1999. `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll99a.ps`.

59. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum, editor, *Proc. FASE 2000 - Fundamental Approaches to Software Engineering*, number 1783 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.

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. The CoFI Task Group on Language Design. CASL The Common Algebraic Specification Language Summary. Version 1.0. Technical report, 1999. Available on `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.

64. G. Winskel and M. Nielsen. Models for Concurrency. Technical Report 492, DAIMI PB, 1992.

65. M. Wirsing. Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoret. Comput. Sci.*, volume B, pages 675–788. Elsevier, 1990.

Contents