

13 Algebraic Specification of Concurrent Systems

Egidio Astesiano¹, Manfred Broy², and Gianna Reggio¹

¹ DISI – Dipartimento di Informatica e Scienze dell’Informazione
Università di Genova – Via Dodecaneso, 35 – Genova 16146 – Italy
{astes,reggio}@disi.unige.it <http://www.disi.unige.it>

² Institut für Informatik der Technischen Universität München
80333 München – Germany
broy@informatik.tu-muenchen.de
<http://wwwbroy.informatik.tu-muenchen.de/>

Introduction

A process is a unit with the capacity of performing an activity by which it may interact with other units and/or with the environment. The interactions may involve communicating, synchronizing, cooperating, acting in parallel, competing for resources with other processes and/or with the environment. By “concurrent systems” we mean processes which may consist of other processes (or in turn concurrent systems) operating concurrently.

Most software systems are concerned with concurrent systems and thus it is of paramount importance to provide good formal support to the specification, design, and implementation of concurrent systems. Algebraic/logic methods have also found interesting applications in this field, especially to treat at the right level of abstraction the relevant features of a system, helping to hide the unnecessary details and thus to master system complexity.

Due to the particularly complex nature of concurrent systems, and contrary to the case of classical (static) data structures, there are different ways of exploiting algebraic methods in concurrency. First of all, we do not have a single satisfactory model and view for processes and concurrent systems, like input–output functions for sequential input–output systems. Hence, algebraic methods need to be applied to different models. Moreover, in the literature, we can distinguish at least four kinds of approaches.

- A1** The algebraic techniques are used at the metalevel, for instance, in the definition or in the use of specification languages. Then a specification involves defining one or more expressions of the language, representing one or more systems. This is, for example, the case in ACP, CCS, and CSP [BK86,Mil89,Hoa85].
- A2** A particular specification language (technique) for concurrent systems is complemented with the possibility of abstractly specifying the (static) data handled by the systems considered using algebraic specifications.

We can qualify the approaches of this kind by the slogan “plus algebraic specifications of static data types”.

- A3** These methods use particular algebraic specifications having “dynamic sorts”, which are sorts whose elements are/correspond to concurrent systems. In such approaches there is only one “algebraic model” (for instance, a first-order structure or algebra) in which some elements represent concurrent systems.

We can qualify the approaches of this kind as “algebraic specifications of dynamic-data types”, which are types of dynamic data (processes/concurrent systems).

- A4** These methods allow us to specify an (abstract) data type, which is dynamically changing with time. In such approaches we have different “algebraic” models corresponding to different states of the system.

We can qualify the approaches of this kind as “algebraic specifications of dynamic data-types”; here the data types are dynamic.

We have organized the paper around the classification above, providing significant illustrative examples for each of the classes. The list of the examples is not exhaustive; moreover, we have given a greater emphasis to the approaches representing an extension to concurrency of algebraic specification techniques. For example, this is why for **A1** we have presented in some detail only CCS, the Calculus of Communicating Systems of R. Milner, as the first and paradigmatic example, though the various versions of CSP, ACP, and the like are of comparable importance as for abundance of literature, theoretical investigations and illustrative applications. Indeed the viewpoint of the process algebra approach is more concerned with formal models of processes via appropriate combinators, in which case the specification problem is handled by adopting a model-oriented approach. The same applies to Petri nets, which represent the earliest attempt (apart from automata) to provide formal models for processes and are as important as CCS, CSP, and the like. Here, within **A2**, we have outlined a formalism concerned with algebraic extensions of Petri nets.

To present a more complete overview, we should also treat another class of approaches, which can be termed “algebraic techniques/tools for dynamics”. These are interesting approaches where technical tools developed in the algebraic field are used formally to capture the dynamic nature of processes. Among them, we can recall the use of the hidden sort algebras and specifications, see, for instance, [GD94], and the use of coalgebras and coalgebraic specifications, see [JR97], also for further references. However we cannot cover these approaches, essentially for lack of space; moreover, coalgebraic methods are quite recent and in full development, compared to those covered in this chapter.

Similarly we do not have space to present other methods, where the usual algebraic specifications of static data types are used instead in a particular clever way to specify processes, see, for instance, [BCPR96].

Particular examples of approaches of the four kinds, presented in Sections 13.1 – 13.4 respectively, are neither a complete list, nor have they been chosen because we think they are the best representatives.

Our rationale has been mainly to present representatives. In particular, there is no intention of providing a comparative study of the methods. This is a goal outside the scope of the book.

The general notions about the specification of concurrency needed for understanding the approaches presented are briefly summarized at the beginning of the various sections.

We use a common example for the presentation of all approaches, a very simple concurrent system consisting of a buffer and a user, informally described below.

The **Bit** example

The system **Bit** (called Bit since it is really very small) consists of two components in parallel: a user and a buffer. The buffer is organized as a queue and contains integers; it may obviously receive and return integer values; it may break down, in which case its content will be 10^{10} , and, moreover, it may happen that the last element of its content is duplicated.

When the system is started by the environment, the buffer is empty and the user puts in sequence 0 and 1 on the buffer; then it gets the first element from the buffer. If this element is the number 0 the user must inform the environment of the correct working of the buffer, otherwise it must signal that there is an error.

Thus **Bit** is an interactive concurrent system with components having both autonomous activities (as the buffer failures) and cooperations (the user writing/reading the buffer), and using some static data (integers); furthermore it also has some relevant static/functional aspects, as the queue organization of the buffer.

Some relevant requirements on **Bit** are:

- R0** The buffer must always be able to receive any integer value.
- R1** When the user is terminated, it cannot perform an activity again.
- R2** In at least one case, the system must behave correctly.
- R3** After being started, it will eventually signal OK or ERROR.
- R4** OK and ERROR are signaled at most once, and it cannot happen that both are signaled.
- R5** The user puts integers on and gets integers from the buffer.

13.1 Process algebras

Process Algebras and Calculi, exemplified by CCS, CSP, ACP, and the like, are the most notable example of the use of algebraic methods in the definition and the use of specification languages (approach **A1**).

Labeled transition systems (abbreviated to lts), as models of processes, underlie CCS and many other variations of process algebras, and are also used in many logical/algebraic specification formalisms. Thus we start this section with the fundamental concepts about lts's and their semantics. Note that the first appearance of lts was in the theory of nondeterministic automata; however, the key idea of using labeled transitions to represent the capabilities of interactions (or participation in events) for describing open systems is generally attributed to Robin Milner in CCS. The related fundamental concept of bisimulation semantics, especially its formalization by maximum fixpoint, is due to David Park.

13.1.1 Modeling processes with labeled transition systems

For the first use of labeled transition systems for the modeling concurrency, see [Mil80,Plo83].

A *labeled transition system* (*lts*) is a triple

$$\langle STATE, LABEL, \rightarrow \rangle,$$

where *STATE* and *LABEL* are two sets, the *states* and *labels* of the system, and $\rightarrow \subseteq STATE \times LABEL \times STATE$ is the *transition relation*. A triple $\langle s, l, s' \rangle \in \rightarrow$ is said to be a *transition* and is usually written $s \xrightarrow{l} s'$.

Given an lts we can associate with each $s_0 \in STATE$ the so-called *transition tree*, that is, the tree whose root is s_0 , where the order of the branches is not considered, two identically decorated subtrees with the same root are considered as a unique subtree, and if it has a node n decorated with s and $s \xrightarrow{l} s'$, then it has a node n' decorated with s' and an arc decorated with l from n to n' .

A process P is thus modeled by a transition tree determined by an lts $\langle STATE, LABEL, \rightarrow \rangle$ and an initial state $s_0 \in STATE$; the nodes in the tree represent the intermediate (interesting) states of the life of P , and the arcs of the tree the possibilities of P of passing from one state to another. It is important to note here that an arc (a transition) $s \xrightarrow{l} s'$ has the following meaning: P in the state s has the *capability* of passing into the state s' by performing a transition, where the label l represents the interaction with the environment during such a move; thus l contains information on the conditions on the environment for the capability to become effective, and on the transformation of such environment induced by the execution of the action.

Concurrent systems, which are processes having cooperating components that are in turn other processes (or concurrent systems), can be modeled through particular lts obtained by composing other lts describing such components.

By associating with a process P the transition tree having root P we give P an operational semantics: two processes are operationally equivalent

whenever the associated transition trees are the same, see [Mil80]. However in most cases such semantics is too fine, since it takes into account all operational details of the process activity. It may happen that two processes which we consider semantically equivalent have associated different transition trees. A simple case is when we consider the trees associated with two deterministic processes interacting with the environment only by returning a final result (e.g., two PASCAL programs) represented by two states p and p' : they only perform internal activities except for the last transitions, and thus the associated transition trees reported below are:

$$\begin{array}{ccccccc} p & \xrightarrow{\tau} & p_1 & \xrightarrow{\tau} & \dots & \xrightarrow{\tau} & p_n \xrightarrow{OUT(r)} p_F \\ p' & \xrightarrow{\tau} & p'_1 & \xrightarrow{\tau} & \dots & \xrightarrow{\tau} & p'_m \xrightarrow{OUT(r')} p'_F \end{array}$$

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

From this simple example, we can also appreciate that we get various interesting semantics on processes modeled by lts depending on what we observe (see, e.g., [Mil80,NH84]). For instance, consider the well-known strong bisimulation semantics of Park [Par81] and Milner [Mil80] and the trace semantics [Hoa85]. In the first case, two processes are equivalent iff they have the same associated transition trees after the states have been forgotten. In the second case, two processes are equivalent iff the corresponding sets of traces (streams of labels), obtained traveling along the maximal paths of the associated transition trees, are the same. In general, the semantics of processes depends on what we are interested in observing.

Now we show how to define precisely strong bisimulation over an lts $\langle STATE, LABEL, \rightarrow \rangle$. A binary relation R on $STATE$ is a *strong bisimulation* iff, for all $s_1, s_2 \in STATE$, if $s_1 R s_2$, then

1. if $s_1 \xrightarrow{l} s'_1$, then there exists s'_2 such that $s'_1 R s'_2$ and $s_2 \xrightarrow{l} s'_2$;
2. if $s_2 \xrightarrow{l} s'_2$, then there exists s'_1 such that $s'_1 R s'_2$ and $s_1 \xrightarrow{l} s'_1$.

The *maximum strong bisimulation* \sim for an lts is defined as the union of all strong bisimulations. We have that \sim is a strong bisimulation and that for all strong bisimulations $R, R \subseteq \sim$.

Similarly we can define weak bisimulation over an lts; in this case the internal transitions, i.e., those corresponding to a null interaction with the environment, are not considered when they have no visible consequence. Technically we use $\tau \in LABEL$ to label internal transitions.¹ We define an auxiliary transition relation

$$\Rightarrow \subseteq STATE \times LABEL \times STATE$$

¹ The symbol τ was used for the first time by Milner for CCS internal transitions, see Section 13.1.2.

as follows:

$$\begin{aligned}
& s \xrightarrow{\tau} s, \\
& \text{if } s \xrightarrow{l} s', \text{ then } s \xrightarrow{l} s', \\
& \text{if } s \xrightarrow{\tau} s' \text{ and } s' \xrightarrow{l} s'', \text{ then } s \xrightarrow{l} s'', \\
& \text{if } s \xrightarrow{l} s' \text{ and } s' \xrightarrow{\tau} s'', \text{ then } s \xrightarrow{l} s''.
\end{aligned}$$

A binary relation R on $STATE$ is a *weak bisimulation* iff, for all $s_1, s_2 \in STATE$, if $s_1 R s_2$, then

1. if $s_1 \xrightarrow{l} s'_1$, then there exists s'_2 such that $s'_1 R s'_2$ and $s_2 \xrightarrow{l} s'_2$;
2. if $s_2 \xrightarrow{l} s'_2$, then there exists s'_1 such that $s'_1 R s'_2$ and $s_1 \xrightarrow{l} s'_1$.

The *maximum weak bisimulation* \approx is the union of all weak bisimulations. We have that \approx is a weak bisimulation and that for all weak bisimulations R , $R \subseteq \approx$.

Example 13.1. (Bit using labeled transition systems) Here we give the modeling the two components of **Bit**, the user and the buffer, and **Bit** itself respectively.

$$USER = \langle STATE_U, LABEL_U, \rightarrow_U \rangle$$

$$STATE_U =$$

$$\{Initial, Putting_0, Putting_1, Reading, Terminated\} \cup \{Read_i \mid i \in \Sigma\}$$

$$LABEL_U = \{START, ERROR, OK\} \cup \{PUT_i, GET_i \mid i \in \Sigma\}$$

\rightarrow_U is graphically represented by depicting the resulting graph in Figure 13.1. Notice that in the state *Reading* the user has infinite action capabilities, one for each possible value that can be obtained from the buffer.

$$BUFFER = \langle \Sigma, LABEL_B, \rightarrow_B \rangle$$

$$LABEL_B = \{RECEIVE_i, RETURN_i \mid i \in \Sigma\} \cup \{\tau\}$$

\rightarrow_B contains the following triples, where $i \in \Sigma$, $q \in \Sigma^*$:

$$\begin{array}{ll}
q \xrightarrow{RECEIVE_i} q \cdot i & i \cdot q \xrightarrow{RETURN_i} q \\
i \cdot q \xrightarrow{\tau} i \cdot i \cdot q & q \xrightarrow{\tau} 10^{10},
\end{array}$$

$$SYSTEM = \langle STATE_S, LABEL_S, \rightarrow_S \rangle$$

$STATE_S$ consists of pairs of states of the buffer and the user.

$$LABEL_S = \{START, \tau, OK, ERROR\}$$

\rightarrow_S contains the following triples, where $i \in \Sigma$, $u, u' \in STATE_U$, $b, b' \in STATE_B$, $\langle \rangle$ is the empty stream:

$$\begin{array}{ll}
\langle \rangle, u \xrightarrow{START} \langle \rangle, u' & \text{if } u \xrightarrow{START} u' \\
\langle b, u \rangle \xrightarrow{\tau} \langle b', u' \rangle & \text{if } b \xrightarrow{RECEIVE_i} b' \text{ and } u \xrightarrow{PUT_i} u', \\
\langle b, u \rangle \xrightarrow{\tau} \langle b', u' \rangle & \text{if } b \xrightarrow{RETURN_i} b' \text{ and } u \xrightarrow{GET_i} u', \\
\langle b, u \rangle \xrightarrow{OK} \langle b, u' \rangle & \text{if } u \xrightarrow{OK} u',
\end{array}$$

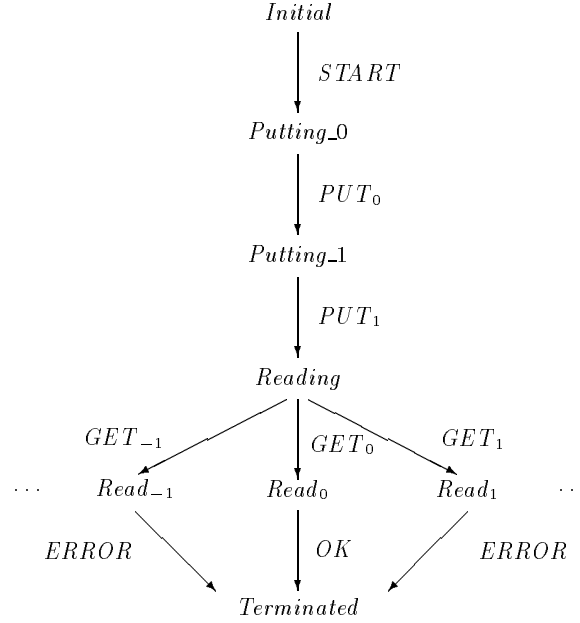


Fig. 13.1. The transitions of the user

$$\langle b, u \rangle \xrightarrow{ERROR}_S \langle b, u' \rangle \quad \text{if } u \xrightarrow{ERROR}_U u',$$

$$\langle b, u \rangle \xrightarrow{\tau}_S \langle b', u \rangle \quad \text{if } b \xrightarrow{\tau}_B b'$$

Notice that *SYSTEM*, defined in a modular way by using *USER* and *BUFFER*, is an example of how we can compose processes operating in parallel. However, if we forget how it has been defined, then we cannot see its concurrent structure. For example, the fact that the transition

$$\langle 0 \cdot 1, Reading \rangle \xrightarrow{\tau}_S \langle 1, Read_0 \rangle$$

corresponds to the synchronous execution of the buffer and user action capabilities, labeled by *RETURN*₀ and *GET*₀ respectively, cannot be deduced by examining *SYSTEM* alone.

13.1.2 Process calculi and algebras

By process calculi and algebras we mean those approaches which specify processes, whose foremost representatives are the many formalisms known under the collective names CCS, CSP, and ACP. The formalisms, though quite different in some fundamental technical aspects, share some basic underlying ideas:

- as in λ -calculi, processes are represented by terms built over a set of combinators concerning all aspects of process behavior, from flow of con-

trol of single processes to operators for composing processes in parallel; the combinators are different, both as a matter of taste and for technical reasons;

- processes are essentially modeled by transition trees;
- the primitive means of interaction between processes is synchronization, that can be interpreted equivalently as synchronization of exchanging data and simultaneous participation in an event;
- emphasis is laid on algebraic laws stating equivalence of processes;
- a concept of refinement is based on containment of behaviors: refining means reducing the amount of possible behaviors.

We present the basic features of CCS, considered a breakthrough in the field, and will briefly comment on ACP and CSP.

CCS, developed by Robin Milner, basically adopts an operational (transition) semantics, associating with each process a transition tree (graph); on the basis of the transition semantics, some equivalences are defined on the processes (various bisimulations and operational equivalences), and laws are proven stating equivalences on processes; the set of laws is usually a complete axiomatization of the semantics over finite processes.

We refer to [Mil89] as a basic reference.

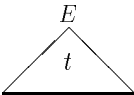
For explanatory purposes, we can start by looking at CCS as a language for describing possibly infinite transition trees.

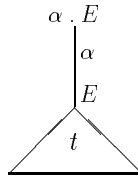
If \mathcal{A} denotes a set of basic names, then $\overline{\mathcal{A}} = \{\overline{a} \mid a \in \mathcal{A}\}$ is the set of the conames and $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$, with $\overline{\overline{l}} = l$. A special label τ indicates the so-called silent action, i.e., an action not visible outside, since it corresponds to a communication taking place within the process; the set of the actions (or, more accurately, capabilities of action), i.e., the labels, is then $\mathcal{ACT} = \mathcal{L} \cup \{\tau\}$, ranged over by α .

First we have the basic combinators for describing finite depth transition trees:

- (1) prefixing $\alpha . E$
- (2) summation $\sum_{i \in I} E_i$, I an indexing set

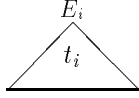
where E denotes a generic CCS expression.

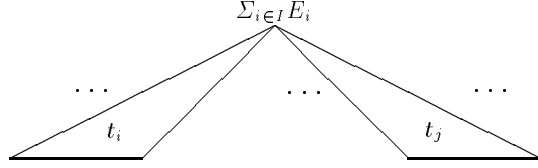
Assume that E represents a tree  with root E , then $\alpha . E$ represents the tree



with root $\alpha . E$. This is formalized by an appropriate semantic clause (inductive rule defining \rightarrow)

$$\mathbf{Act} \quad \frac{}{\alpha . E \xrightarrow{\alpha} E} .$$

Assume that each E_i represents a tree  with root E_i , then $\Sigma_{i \in I} E_i$ represents the tree



The related semantic clause is

$$\mathbf{Sum}_j \quad \frac{E_j \xrightarrow{\alpha} E'_j}{\Sigma_{i \in I} E_i \xrightarrow{\alpha} E'_j} \quad j \in I .$$

In other words clause (1) allows us to describe adding an arc and clause (2) to describe branching. Notice that for $I = \emptyset$, clause (2) defines one expression, also written **nil** or **0**, corresponding to a leaf on a tree.

Infinite depth trees are defined as usual by recursion, say $A =_{\text{def}} P$, where the name of the process A (a constant) may appear in P , which denotes an expression. Of course multiple recursion is possible. The semantics is as usual

$$\mathbf{Con} \quad \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad A =_{\text{def}} P .$$

To handle interaction between processes, a basic combinator for parallelism is introduced:

$$(3) \quad E_1 \parallel E_2$$

There are many ways to define the semantics of the combinator \parallel ; the original one, which we report here, formalizes a notion of synchronization/communication by handshaking communication (**Comm₃**) and of parallel execution by interleaving (**Comm₁**, **Comm₂**).

$$\mathbf{Comm}_1 \quad \frac{E \xrightarrow{\alpha} E'}{E \parallel F \xrightarrow{\alpha} E' \parallel F} \quad \mathbf{Comm}_2 \quad \frac{F \xrightarrow{\alpha} F'}{E \parallel F \xrightarrow{\alpha} E \parallel F'}$$

$$\mathbf{Comm}_3 \quad \frac{E \xrightarrow{l} E' \quad F \xrightarrow{\bar{l}} F'}{E \parallel F \xrightarrow{\tau} E' \parallel F'} \quad l \in \mathcal{L}$$

Rule **Comm₃** says that synchronization may take place whenever the capabilities are complementary (l and \bar{l}).

In CCS we also have two other operations,

$$(4) \quad E/L \quad \text{Restriction}$$

(5) $E[f]$ Relabeling,

where $L \subseteq \mathcal{L}$ denotes a set of nonsilent actions, and f is a function from \mathcal{L} to \mathcal{L} such that $f(\bar{l}) = f(\bar{l})$; f can be extended to \mathcal{ACT} by setting $f(\tau) = \tau$.

The semantics of (4) and (5) is given by

$$\mathbf{Res} \quad \frac{E \xrightarrow{\alpha} E'}{E/L \xrightarrow{\alpha} E'/L} \quad (\alpha \notin L \cup \bar{L}) \qquad \mathbf{Rel} \quad \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}.$$

Notice that relabeling is essentially a user facility for defining processes, with no behavioral meaning, while restriction means hiding from the outside all the action capabilities in L and its complementary set.

The given semantics is operational in nature (though it can also be given in a denotational way) and serves the main purpose of associating a labeled transition system with process expressions.

As we have seen in the preceding section, various semantics can be associated with lts. Depending on the chosen semantics, various laws can be proved about CCS expressions. For example, adopting strong bisimulation semantics, denoted by \sim , the following laws hold ($P + Q = \Sigma\{P, Q\}$)

- (1) $(\alpha . Q)/L = \begin{cases} \mathbf{nil} & \text{if } \alpha \in L \cup \bar{L} \\ \alpha . (Q/L) & \text{otherwise} \end{cases}$
- (2) $(\alpha . Q)[f] = f(\alpha) . Q[f]$
- (3) $(Q + R)/L = Q/L + R/L$
- (4) $(Q + R)[f] = Q[f] + R[f]$
- (5) $P + Q = Q + P$
- (6) $P + (Q + R) = (P + Q) + R$
- (7) $P + P = P$
- (8) $P + \mathbf{nil} = P$

Also a fundamental law, called the expansion law, can be proved, showing that we can eliminate from a process expression the parallel operator, restriction, and relabeling, thus showing the essence of interleaving semantics as reducing parallel execution to nondeterministic choice. A simpler form concerned only with the parallel combinator is as follows:

- (9) $P_1 \parallel \dots \parallel P_n = \Sigma\{\alpha . (P_1 \dots \parallel P'_i \parallel \dots \parallel P_n) \mid P_i \xrightarrow{\alpha} P'_i, 1 \leq i \leq n\} + \Sigma\{\tau . (P_1 \dots \parallel P'_i \parallel \dots \parallel P'_j \parallel \dots \parallel P_n) \mid P_i \xrightarrow{l} P'_i, P_j \xrightarrow{\bar{l}} P'_j, 1 \leq i < j \leq n\}$

It can be shown that strong bisimulation is a congruence for CCS, since it is substitutive under all combinators and recursive definitions. For example, if $P_1 \sim P_2$, then we have

$$\alpha . P_1 \sim \alpha . P_2 \quad P_1 + Q \sim P_2 + Q, \text{ etc.}$$

Also, under reasonable conditions, recursive definitions uniquely identify a process modulo its bisimulation.

Unfortunately, passing to weak bisimulation (observation equivalence in [Mil89]), we do not get a congruence any longer and thus another equivalence is introduced, called equality (or observation congruence), which implies weak bisimulation. It can be shown that, since CCS is as powerful as Turing machines, no effective axiomatization of equality exists. However the laws provide effective axiomatization for smaller classes of processes, such as the finite processes.

Our presentation shows the role of laws in CCS, as derived theorems from essentially operational semantics. A somewhat different approach has been taken by ACP, mainly developed by Bergstra, Klop, and Baeten (see [BW90] also for references). There the starting point is a complete axiomatization (usually by equations or conditional axioms) of some equivalence (e.g., strong or weak bisimulation) for finite processes; thus two finite processes are equivalent iff their equality can be proven by equational/conditional deduction. Then recursion is added and semantics is again given in terms of graphs, labeled transition systems, or projective limits.

The approach is highly hierarchical, introducing laws for new combinators in a conservative way. Some of the combinators introduced in ACP are due to the technical needs for obtaining complete axiomatizations.

Different again is the CSP approach [Hoa85], where the semantics is denotational and the laws are derived from semantics, and used for reasoning about correctness. The denoted values are different, depending on the richness of the combinators; they range from sets of traces to the so-called refusal sets.

A common problem with the process algebra/calculi approaches is the enormous variety of possible meaningful semantics and thus of the associated derived laws; in one paper [vG90] Van Glabbeek analyzes from a modal logic unifying viewpoint, as many as 155 different semantics.

Example 13.2. (Bit using CCS) The process corresponding to the user is defined by

$$USER = START.\overline{PUT}_0.\overline{PUT}_1.(GET_0.\overline{OK}.\mathbf{nil} + \Sigma_{i \in -\{0\}} GET_i.\overline{ERROR}.\mathbf{nil});$$

the process buffer is defined by the following, mutually recursive definitions

$$\begin{aligned} BUFFER_{\langle \rangle} &= \Sigma_{i \in \{0,1\}} PUT_i.BUFFER_i + \tau.BUFFER_{1010} \\ BUFFER_{i_1 \dots i_k} &= \Sigma_{i \in \{0,1\}} PUT_i.BUFFER_{i_i_1 \dots i_k} + \overline{GET}_{i_k}.BUFFER_{i_1 \dots i_{k-1}} + \\ &\quad \tau.BUFFER_{1010} + \tau.BUFFER_{i_1 i_1 \dots i_k}, \quad i_1 \dots i_k \in \{0,1\}^+ \end{aligned}$$

Finally the system is the parallel composition of the two processes above (initially the buffer is empty)

$$SYSTEM = (BUFFER_{\langle \rangle} \parallel USER) / \{PUT_i, GET_i \mid i \in \{0,1\}\}.$$

The example illustrates the use of CCS in the specification phase, which follows a model-oriented approach: with the help of the CCS language a process is described and then a class of models is defined corresponding to the equivalence class of the process (w.r.t. some equivalence).

13.2 Algebraic specification of static data types

In this section we briefly present some specification techniques following the approach **A2** to the algebraic specification of concurrent systems, that is, approaches integrating a formalism for the concurrent aspects with algebraic specifications of the static data types.

13.2.1 Process calculi plus algebraic specification of data types

In this subsection we briefly present two specification formalisms, LOTOS and PSF, designed following approach **A2**, where the processes are defined by a process–algebra style calculus. The differences between LOTOS and PSF are in the formalism for the algebraic specification part (ACT ONE [EFH83] and ASF [BHK89], respectively) and in the combinators of the process calculus chosen (inspired by those of CCS and ACP, respectively, see Section 13.1).

Process specification formalism (PSF). PSF [MV89,MV90] is the process specification formalism developed by Mauw and Veltink as a base for a set of tools to support the process algebras. The main goal in the design of PSF was to provide a specification language with a formal syntax similar to the process algebra ACP [BW90, Section 4] but also with a notion of data type; to this end ASF (the Algebraic Specification Formalism of [BHK89]) has been incorporated.

The basic specification formalism is equational logic with total algebras. The theory and language of ASF are adopted for handling modular and parameterized specifications.

A PSF specification consists of a series of modules, divided into data modules and process modules. Data modules are algebraic specifications with initial semantics. Process modules are ACP specifications of processes. Formally, a process module consists of

- declarations of the operation symbols for actions and processes (which may have the static data as arguments),
- explicit definitions of the synchronization among such actions,
- process definitions of the form $P(x_1, \dots, x_n) = \text{ACP-expression}$, in which the operators like “+”, “||”, “;”, “**hide**” and “**encaps**”, elementary processes, pure atomic actions, and also P (thus allowing recursive definitions) may appear.

Processes are particular data structures obtained by a given (equational) axiomatization which determines a particular semantics over these structures, embodying ideas of concurrency. This is best understood by looking at the hidden basic concurrent models behind process algebra, which are lts as in CCS and many other approaches; then the axioms provide semantics like strong, trace, or bisimulation semantics and others, see Section 13.1.1. The

hidden model is made evident in some presentations of PSF, where ACP processes are described by means of lts. In any case, since ACP essentially provides a language schema for processes, it is irrelevant, other than for building the tools, how its semantics is given, either by equations or by labeled transitions plus semantic equivalences.

It is instead important to note that in PSF:

- the synchronization of actions can be defined explicitly in the communication part; as a consequence, the synchronization mechanism is not fixed and is parameterized;
- the execution mode is interleaving.

The interface between processes and data types is as follows:

- the atomic actions may have as components some values of the specified data types;
- it is possible to define recursively families of processes indexed on the elements of some sort;
- an infinitary nondeterministic choice indexed on the elements of a sort is available.

The semantics of the data part is a classical algebraic semantics by initiality; the semantics of processes is strong bisimulation, which gives a congruence over the term algebra. Thus the semantics identifies an isomorphism class of structures, as for a data type.

The data part is strictly separated from the process part. Thus it is an **A2** approach; but the concurrent structure here is also specified algebraically, though with a fixed set of primitives parameterized on the actions and the synchronization structure. The result is a completely algebraic specification to which all the techniques and results of ASF can be conveniently applied.

Particularly powerful are the modularization mechanisms in PSF, which are borrowed from ASF but truly deal with the integration of data types and processes; the module concept also supports the import and export of processes and actions.

There is a vast literature on the use of process algebras, with a detailed treatment of classical examples and correctness proofs for implementation. However, these examples should not be confused with applications of a specification method like PSF, which have indeed been introduced for supporting industrial applications. Clearly PSF is applicable to a wide range of significant cases in practice, see, for instance, [MV93], but we see a limitation in its strict policy of message passing and no provision for data sharing. In many cases some amount of coding is required which is not in the spirit of abstract specifications. The same remark applies to execution modes other than interleaving, which have to be simulated by appropriate use of synchronization and restriction mechanisms.

PSF has been devised as a basis for the development of a toolset (see, e.g., [MV89,MV91,PSF97]); in particular, a simulator, a term rewriting, and a proof assistant have been implemented.

LOTOS. LOTOS was probably the first internationally known (since 1984), algebraic specification formalism for concurrency [BB87,I.S89]; most importantly, it is an official ISO specification language for open distributed systems, a qualification which alone would rank it high in an ideal value scale of possible important applications. However, LOTOS is interesting also because it represents an early paradigm of which PSF can be considered an improvement. Because of this, we do not go into a detailed discussion of LOTOS; it is enough to compare it with PSF to understand its structure.

LOTOS adds classical algebraic specifications into a language for concurrency like PSF; but it uses ACT ONE [EFH83] instead of ASF and a process description based on an extension of CCS with several derived combinators (e.g., input/output of structured values, sequential composition with possible value passing, enabling/disabling operators) instead of the process algebra ACP. The basic specification formalism (equational logic with total algebras) and process bisimulation semantics are the same.

PSF is an improvement over LOTOS (see a discussion in [MV89]), since it allows more freedom in the definition of synchronization mechanisms and supports import/export of action/processes, thus becoming more flexible for stepwise development.

Throughout these years LOTOS has been used in several practical applications and nowadays tools for helping to write correct LOTOS specifications have been developed (see, for instance, the ESPRIT project LOTOSHERE [vE91]). Recently a new, revised version of LOTOS (E-LOTOS, for Enhancement to LOTOS) has been developed and presented as a standard [LOT97], taking into account the needs that emerged through its application; enhancements concern the data part (built-in, partial operations), the concurrency part (noninterleaving semantics, real time, priorities), and the whole organization of the specifications (introduction of modules).

Example 13.3. (Bit using LOTOS) The data part is given by the specification *INT_QUEUE*, shown in Appendix A, and by the following:

```
spec MESSAGE =
  sorts  message      ** messages exchanged with the environment
  opns  OK, ERROR: → message
```

Bit is given as the parallel composition of two processes corresponding to the buffer and the user.

The gates of such processes and their connections are graphically represented in Figure 13.2.

Below “?” and “!” distinguish input/output actions, “;”, “... → ...”, “[]”, “[|]”, and “i” denote respectively action prefixing, Boolean guards, non-deterministic choice, parallel combinator and internal action.

In the definition of *BUFFER*, *Put* and *Get* are the gates and *q* is a process parameter of sort *queue*.

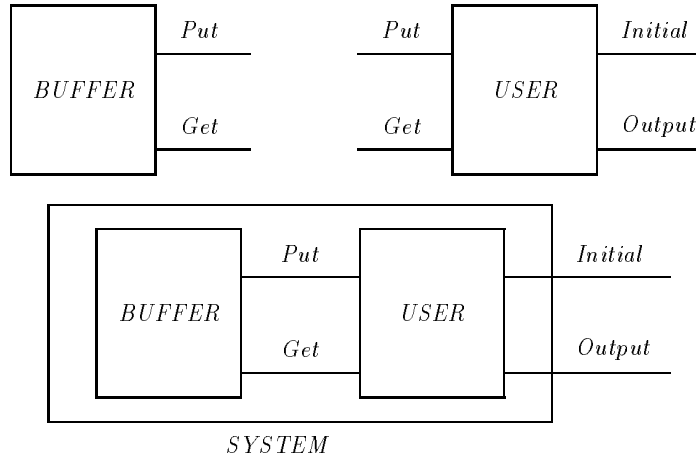


Fig. 13.2. Structure of the LOTOS specification of **Bit**

```

process BUFFER[Put, Get](q : queue) :=
  [Not_Empty(q)] → Get! First(q); BUFFER[Put, Get](Remove(q)) []
  Put? x : int; BUFFER[Put, Get](Put(x, q)) []
  i; BUFFER[Put, Get](Put(1010, Empty)) []
  i; BUFFER[Put, Get](Dup(q))
end process

process USER[Initial, Put, Get, Output] :=
  Initial?; Put!0; Put!1; Get? x : int;
  ([not(x = 0)] → Output! ERROR) []
  [x = 0] → Output! OK)
end process

process System[Initial, Output] :=
  BUFFER[Put, Get](Empty) ||| USER[Initial, Put, Get, Output]
end process

```

Note the similarity between LOTOS and CCS.

13.2.2 Petri nets

Petri nets are among the earliest and most influential models of concurrency.

Net models. Here we briefly present (elementary) nets, the basic models for the various specification formalisms generally called “Petri nets”; all variants arise either by putting some restrictions on the allowed nets, adding minor features, or offering more compact/simple ways to present the nets

(e.g., colored nets, high-level nets, . . .); in some cases, a slightly different terminology is used. [Rei85,RT86] give general overviews and further references for net models, while [Rei98] is more concerned with the use of nets for specifying concurrent systems.

A *net* N is a triple $\langle P, T, F \rangle$, where P and T are two sets and F is a subset of $(P \times T) \cup (T \times P)$; P , T , and F are respectively called the sets of the *places*, the *transitions*, and the *arcs* of the net; F is also called the *flow relation*.

Nets are graphically represented by depicting places, transitions, and arcs respectively as circles, boxes, and arrows, see, e.g., Figure 13.3.

Example 13.4. (Bit using Petri nets) A Petri net modeling **Bit** is reported in Figure 13.3; note that to make the drawing small enough we have assumed that the buffer may only fail when its content is 01.

The behavior of a net is defined as follows.

Given a transition $t \in T$, we define $\cdot t = \{p \mid p \in P, \langle p, t \rangle \in F\}$ (the *preconditions* of t) and $t \cdot = \{p \mid p \in P, \langle t, p \rangle \in F\}$ (the *postconditions* of t).

Any function s from P into \mathbb{N} is called a (*global*) *state* (or *marking*) of N ; graphically represented by putting $s(p)$ \bullet 's (called *tokens*) on the place p , for any $p \in P$; the net in Figure 13.3 is in a state characterized just by one token on each of the places *Initial* and *Empty*.

A transition $t \in T$ is *enabled* in a state s (or *may fire*) iff, for any $p \in \cdot t$, $s(p) > 1$. If t is enabled in s , then

$$s' = \lambda p. \begin{cases} s(p) - 1 & \text{if } p \in \cdot t \\ s(p) + 1 & \text{if } p \in t \cdot \\ s(p) & \text{otherwise} \end{cases}$$

is called the *successor state of s with t* (or *the state obtained after the firing of t in s*) and $s \xrightarrow{t} s'$ is called a *step* in N .

A *system net* (*s-net* for short) is a net N , together with a state, called the *initial state*.

The single steps of an s-net can be composed in runs: $\langle s_i \xrightarrow{t_i} s_{i+1} \rangle_{i \in I}$ with s_0 the initial state and $I = \{0, \dots, n\}$ ($I = \mathbb{N}$) is a *finite (infinite) interleaved run*.

$$\begin{aligned} & \text{Empty} : 1; \text{Initial} : 1 \xrightarrow{\text{START}} \text{Empty} : 1; \text{Putting}_0 : 1 \xrightarrow{\text{INT1}} \\ & 0 : 1; \text{Putting}_1 : 1 \xrightarrow{\text{INT2}} 01 : 1; \text{Reading} : 1 \xrightarrow{\text{INT3}} 01 : 1; \text{Ok} : 1 \xrightarrow{\text{BREAK}} \\ & 10^{10} : 1; \text{Ok} : 1 \xrightarrow{\text{OK}} 10^{10} : 1; \text{Terminated} : 1 \end{aligned}$$

is an example of an interleaved run of the net in Figure 13.3, where a state is represented by listing the places with the number of their tokens, forgetting those without tokens.

Some occurrences of transitions in an interleaved run that are seen as ordered one after the other may be independent. Thus they may also have

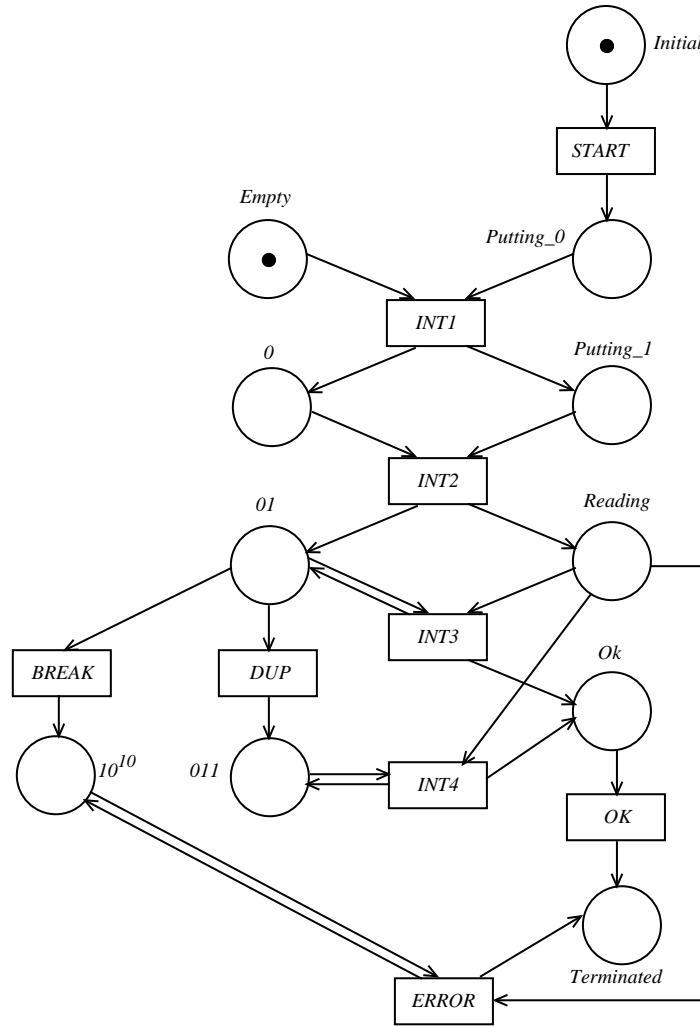


Fig. 13.3. A Petri net modeling Bit

happened in the reverse order; in the run depicted above, the two last transitions are independent. We give another definition of a run, making explicit the concurrent aspects.

Concurrent runs are represented by special nets; the underlying idea is that a concurrent run of N is a net, whose places and transitions are labeled with the places and transitions of the original net N and correspond to their occurrences and firings in the run, and where the flow relation corresponds to the causal relationships among them.

Due to lack of space we skip the complete definition of concurrent runs, and just give in Figure 13.4, as an example, the concurrent run corresponding to the run above; here we can see how the two last transitions are not causally related.

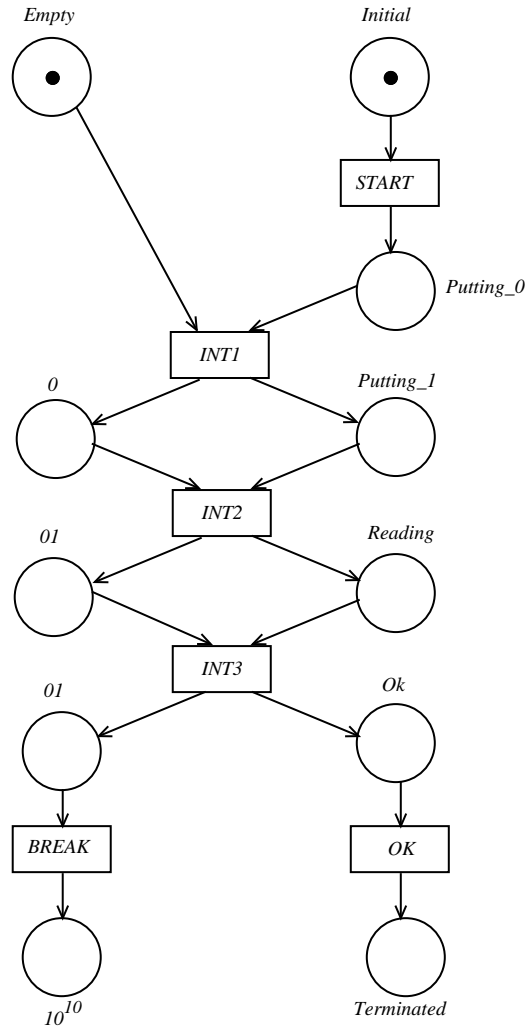


Fig. 13.4. A concurrent run of the net modeling **Bit**

The features of concurrent systems that can be nicely handled by modeling them with nets are as follows:

- local and global states abstractly represented as token distributions over the places;
- atomic actions (the transitions);
- (local) causality and effect between actions and states;
- mutual independence/causality of actions;

while there is no acceptable way to model

- the (distributed) structure of the system;
- the modular decomposition of systems;
- open/interacting/reactive systems (no distinction between internal and external transitions/places: in the net in Figure 13.3, *START*, *OK*, and *ERROR* should be the external transitions), and therefore no way to define modularly the net corresponding to a complex system by putting together the nets corresponding to its components.

Clearly many new formalisms extending nets with other features have been proposed in the literature overcoming some of these problems.

Nets are particularly apt for analyzing the modeled systems, formally checking whether the system has or does not have some properties, including both safety and liveness properties (see Section 13.2.3); several techniques have been provided for that, one of the most relevant is based on “place-invariants”.

A *place-invariant* is a linear combination (summation) with (also zero) integer coefficients of the number of tokens contained in the places, which is not changed by the firing of the transitions; all place-invariants of a net may be found by solving a linear system of equations with integer coefficients, thus it is possible to have software tools for finding them.

For example, two invariants of the net in Figure 13.3 are:

$$1 \cdot \textit{Initial} + 1 \cdot \textit{Putting}_0 + 1 \cdot \textit{Putting}_1 + 1 \cdot \textit{Reading} + 1 \cdot \textit{Ok} + 1 \cdot \textit{Terminated} = 1$$

$$1 \cdot \textit{Empty} + 1 \cdot 0 + 1 \cdot 01 + 1 \cdot 011 + 1 \cdot 10^{10} = 1$$

(corresponding to saying that the user and the buffer are always in one and only one state); while

$$1 \cdot \textit{Ok} - 1 \cdot 01 = 0$$

is not an invariant (the user may be in the Ok state while the buffer content is not 01).

High-level algebraic Petri nets. Nets, as presented in Section 13.2.2, allow us to model several aspects of concurrent systems. However, if we want to use them for significant examples we have to handle very large (if not infinite) nets; e.g., consider the Petri net modeling a system using integer numbers. Moreover in practical applications we have also to handle complex data structures.

To fix the first kind of problem, the basic nets have been extended in several ways: a transition firing can test for the presence of, delete, and add finite sets of tokens (creating Petri nets as originally defined). Later on, the unique black token was replaced by colored tokens, producing the so-called “Colored nets”; here the firing of a transition also depends on the colors of the tokens present in the places, and its firing deletes and adds sets of colored tokens. More generally, tokens can be considered to be data taken in some data structure; this leads to the so-called “algebraic nets” where such a data structure is given by a (many-sorted) algebra.

A richer structure of the tokens allows us to introduce “high-level” nets, where the basic idea is that the arcs are decorated with symbolic expressions, describing in a compact way the sets of tokens causing the firing of a transition, and those deleted and added by such firing. Moreover, now the transitions may also be decorated by some expression corresponding to performing some checks on the tokens present in the places in the premise of the transition, for instance, putting in relation the color of the token in a place with that of the token in another place (the token of type natural in one place should be the length of the token of type queue in another). In the literature it is possible to find several proposals for high-level algebraic nets (see, for instance, [Rei91,BCM88,DH91,Vau87]). The reader interested in a more detailed study of such nets may, for instance, consult [JR91]; here we just briefly present their basic features.

A *high-level algebraic net system* consists of:

- a net $N = \langle P, T, F \rangle$ (*the schema*);
- a signature Σ and a set of sorted variables X ;
- an association with each place in P of a sort of Σ (places are typed with the sorts);
- an association with each arc in F of a set of terms built on Σ and X having the sort of the place source or target of the arc (*arc inscriptions*);
- an association with each transition in T of a first-order (conditional, . . .) formula built on Σ and X , where only the variables appearing in the inscriptions of the arcs entering in the transition may appear (*transition inscriptions*);
- a Σ -algebra A (*the data part*);
- an association with each place p in P of a set of elements of $|A|_s$, where s is the sort associated with p (*initial state*).

In some approaches, sets are replaced by multisets, and in others the signatures and the algebras are extended to have sorts and operations for handling sets/multisets of elements of the original sorts; thus arcs are inscribed by terms of sort $set(s)/mset(s)$ and the initial marking consists of elements of the carriers of these set/mset sorts. The algebras used in algebraic nets may be of whichever kind; e.g., there are approaches using homogeneous total algebras and others using many-sorted partial algebras.

The abstraction level of high-level algebraic nets may be enhanced by abstractly giving the data part as an algebraic specification with initial semantics.

Example 13.5. (Bit using algebraic high-level nets) Here we give a specification of **Bit** which is an improvement on that presented in Figure 13.3; the buffer is now organized as a queue and contains integer numbers, and so it is possible to check whether its first element is 0. Notice that the corresponding non-high-level net is infinite.

The data part is given by the following algebraic specification with initial semantics

```
spec DATA =
  enrich INT_QUEUE by
  sorts   token
  opns   •: → token
```

The place *Buffer* has sort *queue*, while *Initial*, *Putting_0*, *Putting_1*, *Reading*, *Ok*, and *Terminated* have sort *token*.

The net is shown in Figure 13.5.

Like classical Petri nets, high-level algebraic nets suffer from their lacking modularity.

13.2.3 Temporal logic

In the field of concurrency, specifications following an axiomatic, or, better, property-oriented style, have been widely used, in general to give the formal specification of the requirements on a concurrent system. In these cases, a specification is just a set of formulas of some logic expressing the requirements on the specified system; among the commonest and most relevant requirements, we have:

1. *liveness properties*: (under some condition) something good will happen eventually in the system; for instance, the system will eventually react to the reception of some stimuli/after that some situation has been reached;
2. *safety properties*: (under some condition) something bad will never happen in the system; for instance, after receiving some stimuli/reaching some situation, some (incorrect) output will not be produced/some (incorrect) situation will be not reached;
3. *fairness properties*: the repeated choice between two alternative activities of the system must be fair (i.e., it cannot happen that one of the two alternatives will be chosen forever); e.g., in the case of two processes trying to access a shared resource, it cannot happen that only one will succeed;

First-order logic is not sufficient to express properties such as those above in full generality: some properties are related to the evolution of the system over time (2), others related to the possible activities of a system (1), or of the components of a system. Following the way indicated by Pnueli in a landmark paper [Pnu77], in recent years various modal/temporal logics have been widely and satisfactorily used to express properties like (1) and (2), or special variants particularly tailored for the system model chosen have been developed (see, e.g., [CR97,Mil89]). Concerning properties such as (3, 4), no fully satisfactory proposal has yet been found (some initial attempts are in [Reg91,ES95]).

Here we only briefly introduce the basic (linear and branching) temporal logics, and afterwards we show how first-order temporal logic with equality may be used to give requirement specifications of concurrent systems where the static data are specified by loose algebraic specifications, following an **A2** approach; in Section 13.3, we present an alternative approach based on temporal logic of kind **A3**.

Variations of temporal logics. Here we briefly recall the definition of a linear/branching temporal logic and give some examples of use of its formulas to express requirements on concurrent systems; for references, see [Sti92, Pnu86,Eme90].

A propositional linear temporal logic. TL is a propositional linear temporal logic with a minimal set of combinators.

Assume that Q is a set of propositional symbols; thus the *formulas* of TL are defined by:

$$\phi ::= Q \mid \phi_1 \Rightarrow \phi_2 \mid \neg \phi \mid \phi_1 \mathcal{U} \phi_2$$

The *models* of TL are sequences of states $\mathcal{M} = \langle s_i \rangle_{i \in \mathbb{N}}$, where a state s is a function from Q into $\{T, F\}$, the set of the truth values.

The *validity* of a formula ϕ over a model $\mathcal{M} = \langle s_i \rangle_{i \in \mathbb{N}}$ is defined as follows:

$$\mathcal{M} \models \phi \quad \text{iff} \quad i, \mathcal{M} \models \phi \text{ for all } i \in \mathbb{N},$$

where

- $i, \mathcal{M} \models Q$ iff $s_i(Q) = T$
- $i, \mathcal{M} \models \phi_1 \Rightarrow \phi_2$ iff $i, \mathcal{M} \models \neg \phi_1$ or $i, \mathcal{M} \models \phi_2$
- $i, \mathcal{M} \models \neg \phi$ iff $i, \mathcal{M} \not\models \phi$
- $i, \mathcal{M} \models \phi_1 \mathcal{U} \phi_2$ iff there exists $j \geq 0$ such that for all h , $i < h < j$, $h, \mathcal{M} \models \phi_1$ and $j, \mathcal{M} \models \phi_2$.

The term “linear” means that in this case the behavior of a system is modeled by a set of executions represented by linear sequences of states, and thus at a given instant state, there is exactly one successor state.

When TL is used to specify a system, we have that the models and the formulas represent respectively the executions of that system and the properties on such executions. For example, $\phi_1 \mathcal{U} \phi_2$ corresponds to saying that in any execution the property represented by ϕ_1 holds until the property represented by ϕ_2 holds, and that ϕ_2 surely will hold. Thus a set of TL formulas Π could be used to specify the requirements on a system: Π determines the class of all systems whose possible executions are included in the class of the models of Π .

\mathcal{U} is the basic combinator; many others suitable for expressing further relevant properties can be derived; among them:

- **true**, **false**, \vee , \wedge , and \Leftrightarrow , defined in the usual way
- $\phi \stackrel{\text{def}}{=} \mathbf{true} \mathcal{U} \phi$ (eventually the property represented by ϕ will hold)
- $\phi \stackrel{\text{def}}{=} \neg \neg \phi$ (the property represented by ϕ will hold forever)
- $\phi_1 \mathcal{WU} \phi_2 \stackrel{\text{def}}{=} (\phi_1 \mathcal{U} \phi_2) \vee \neg \phi_1$ (the property represented by ϕ_1 holds until ϕ_2 will hold, but it is not required that ϕ_2 will eventually hold).

A propositional branching temporal logic. **BTL** is a propositional branching temporal logic with a minimal set of combinators given, following a CTL style. The term “branching” means that in this case the behavior of a system is modeled by a tree whose nodes are decorated by states, and thus at a given instant there may be several different successor states.

As before, assume that Q is a set of propositional symbols; then the *formulas* of BTL are the following, where Δ is the combinator for “quantifying over paths”:

$$\phi ::= Q \mid \Delta \phi \mid \phi_1 \Rightarrow \phi_2 \mid \neg \phi \mid \phi_1 \mathcal{U} \phi_2.$$

The *models* of BTL are transition systems or Kripke structures, where a function from Q into the set of the truth values is associated with each state. Precisely, a model \mathcal{M} is a triple $\langle STATE, \rightarrow, v \rangle$, where $STATE$ is a set, $\rightarrow \subseteq STATE^2$ and v is a function from $STATE$ into the set of the functions from Q into $\{T, F\}$.

The *validity* of a formula ϕ over a model \mathcal{M} is defined as follows.

First we define the set of paths over \mathcal{M} :

$$PATH(\mathcal{M}) = \{ \langle s_i \rangle_{i \in \mathbb{N}} \mid \forall i \in \mathbb{N}. ((s_i \rightarrow s_{i+1} \vee (\forall j. j > i \Rightarrow (s_j = s_i \wedge \nexists s. s_i \rightarrow s))) \}$$

Given $\sigma = \langle s_i \rangle_{i \in \mathbb{N}} \in PATH(\mathcal{M})$ and $h \geq 0$, $\sigma|_h$ denotes the path $s_h \ s_{h+1} \ s_{h+2} \ \dots$.

$$\mathcal{M} \models \phi \quad \text{iff} \quad \sigma, \mathcal{M} \models \phi \quad \text{for all } \sigma \in PATH(\mathcal{M}),$$

where

- $\sigma, \mathcal{M} \models Q \quad \text{iff} \quad v(s_0)(Q) = T$

- $\sigma, \mathcal{M} \models \phi_1 \Rightarrow \phi_2$ iff $\sigma, \mathcal{M} \models \neg \phi_1$ or $\sigma, \mathcal{M} \models \phi_2$
- $\sigma, \mathcal{M} \models \neg \phi$ iff $\sigma, \mathcal{M} \not\models \phi$
- $\sigma, \mathcal{M} \models \phi_1 \mathcal{U} \phi_2$ iff there exists $j \geq 0$ such that for all $h, 0 < h < j$, $\sigma|_h, \mathcal{M} \models \phi_1$ and $\sigma|_j, \mathcal{M} \models \phi_2$
- $\sigma, \mathcal{M} \models \Delta \phi$ iff $\sigma', \mathcal{M} \models \phi$ for all $\sigma' \in PATH(\mathcal{M})$ such that $s_0 = s'_0$.

When **BTL** is used to specify a system, we have that a model represents the whole behavior of such a system, i.e., all its possible executions and at which point the nondeterministic choices are made, and then the formulas represent properties on such behavior. For example, $\neg \Delta (\mathbf{true} \mathcal{U} \phi)$ corresponds to saying that it is not true that in any case the executions of the system will eventually satisfy the property represented by ϕ (if ϕ corresponds to failing, then the formula requires that the system has at least an execution without failures). Thus a set of **BTL** formulas Φ could be used to specify the requirements on a system: Φ determines the class of all systems whose behavior is described by an element of the class of the models of Φ .

Δ is the basic branching combinator; many others that are suitable to express further relevant system properties can be derived; among them

$$\nabla \phi =_{\text{def}} \neg \Delta \neg \phi$$

(at least in one case, i.e., the property represented by ϕ holds in at least one path).

The derived combinators for the path formulas, ∇ and Δ , can be defined as for the linear-time logic.

Further temporal logics. In the previous paragraphs we have briefly sketched two simple logics. In the literature and in the “practice” of specification of concurrent systems, a large number of variants have been proposed; the differences are related to:

anchored version In the model a state (set of states) is singled out to be initial, determined by a special propositional symbol, and the validity of a formula is changed to hold only on such states (paths starting from such states). Formally, for the branching-time case, assume that $s' \in STATE$ is the initial state, thus

$$\mathcal{M} \models \phi \text{ iff } \sigma, \mathcal{M} \models \phi \text{ for all } \sigma \in PATH(\mathcal{M}) \text{ such that } s_0 = s'.$$

edge formulas The models, instead of just being sequences (trees) of states, allow the labeling of the transitions from a state to another; thus they are sequences of states and labels or trees where the arcs are labeled (labeled transition systems). Clearly, the formulas are extended to include “edge formulas” expressing conditions on the next label (see, e.g., [Lam83, CR97]).

first-order The basic formulas, instead of being propositional, are first-order. Now there is the problem of the evaluation of the variables appearing in a formula; usually the symbols appearing in a formula (operations, predicates, variables) are classified into: *rigid*, whose interpretation does not depend on the state where the formula is evaluated, and *flexible*, whose interpretation depends on the state where the formula is evaluated. Consequently a model consists of a standard first-order structure, a variable evaluation (for interpreting the rigid symbols) and a sequence (tree) of states, where with each state is associated a first-order structure and a variable evaluation, for interpreting the flexible symbols. Clearly the carriers of such structures must coincide with those of the structure used for rigid symbols (no sensible and usable proposal is available for overcoming this restriction).

Temporal logic and algebraic specifications (A2). First-order temporal logics allow us to give specifications of concurrent systems, where the properties on the dynamic activity are given using the temporal combinators, while the involved (static) data structures are specified by first-order loose algebraic specifications. The sort symbols plus the rigid symbols give the signature of the data structure, while the flexible symbols describe the states of the system (notice a similarity with the dynamic data-type approach in Section 13.4).

Example 13.6. (Bit using temporal logic (requirements)) In this case we try to give some requirements on **Bit** (see p. 3), instead of specifying its design, as done in previous examples.

The static data structure is now specified by the following loose algebraic specification, where we do not fix the policy followed by the buffer for storing values (e.g., as a queue or as a stack).²

```
spec INT_BUNCH =
  enrich INT by
  sorts   bunch
  opns   Empty: → bunch
         Put: int × bunch → bunch
         First: bunch → int
         Remove: bunch → bunch
  preds  Not_Empty: bunch
         Is_In: int × bunch
  axioms Empty and Put are generators for bunch
         ¬ Is_In(i, Empty)
         Is_In(i, Put(i', b)) ⇔ (i = i' ∨ Is_In(i, b))
```

² In this chapter, for simplicity, we omit the universal quantifiers when writing the specification axioms; thus, e.g., $\neg \text{Is_In}(i, \text{Empty})$ stands for $\forall i: \text{int}. \neg \text{Is_In}(i, \text{Empty})$.

$$\begin{aligned}
\text{Not_Empty}(b) &\Leftrightarrow \exists i. \text{Is_In}(i, b) \\
\text{Is_In}(\text{First}(b), b) & \\
\text{Is_In}(i, \text{Remove}(b)) &\Rightarrow \text{Is_In}(i, b)
\end{aligned}$$

The states of the system are then characterized by the following flexible symbols:

```

opns  Buf_Cont: → bunch    ** the buffer content
preds Putting: int
        ** is the user putting a given integer in the buffer ?
        Reading: int
        ** is the user reading a given integer from the buffer ?
        Terminated:
        ** has the user terminated its activity ?
        Error:
        ** has something erroneous happened in the system ?

```

The following formulas express requirements on **Bit**.

$$\text{Putting}(i) \Rightarrow \Delta \quad (\text{Is_In}(i, \text{Buf_Cont}) \vee \text{Error})$$

If the user is putting i in the buffer, then in any case, eventually, either i will be in the buffer or something erroneous will happen (**R5**).

$$\Delta \quad (\text{Reading}(i) \Rightarrow i = \text{First}(\text{Buf_Cont}))$$

or equivalently

$$\text{Reading}(i) \Rightarrow i = \text{First}(\text{Buf_Cont})$$

In any case, always, if the user is reading i , then i is the first element of the buffer (**R5**).

$$\text{Terminated} \Rightarrow \text{Terminated}$$

Once the user has terminated its activity, it cannot restart (**R1**).

$$\nabla \quad \neg \text{Error}$$

There always exists a possible “correct” behavior (**R2**).

There is no way to express (**R0**), since we cannot express the buffer capabilities of interacting with the user within the system.

13.2.4 Streams and data flow

A model of concurrency where the data structures representing the flow of interaction are made explicit is data flow concepts based on streams. Of course, these temporal formulas do not specify all the interesting properties of a buffer; rather they specify a subset. Temporal logic is not well suited for comprehensive specifications. Therefore it is better to combine it with methods that are more appropriate for specifying safety properties.

Data flow models. For references, see, for instance, [Bro87,Bro97,Bro93,Bro96].

Data flow models of systems are often represented by data flow graphs (also called data flow diagrams). A *data flow graph* is a directed graph, the nodes of which are called data flow nodes and the arcs of which are called data flow arcs. Some arcs may have no sources. These are called *input arcs*. Others may have no target. These are called *output arcs*.

Data flow models are used in many methods in software engineering. They provide a structural view of a system by representing the computing agents by data flow nodes and their communication interconnection by the arrows connecting them. Although data flow diagrams are used in nearly all methods (SA, SADT, SSADM, OMT, SDL, etc.), as well as in many books on operating systems, their meaning is often not well defined and leads to many misinterpretations. The stream model can help to provide a precise meaning for data flow graphics.

There are a number of variations of data flow models. In acyclic data flow models often only one data value is associated with each data flow arc. The data flow nodes are then functions that receive their arguments on their input arcs (one on each arc) and produce one result on each of their output arcs. The data flow diagram accordingly shows a computation tree or a computation graph. This is related to the single assignment languages.

In more sophisticated data flow graphs, we associate a stream of data elements with each data flow arc. This leads to Kahn networks [Kah74]. In the deterministic case each node is associated with a stream processing function that receives its argument streams on its input arcs (one at each arc) and produces one result stream on each of its output arcs. These graphs may be cyclic. This leads to cyclic (recursive) definitions for the streams associated with the arcs. A simple mathematical model for data flow diagrams can be obtained by *stream processing functions*. Nondeterministic data flow diagrams can be handled by associating sets of functions with each node.

The idea of data flow was heavily influenced by the concept of Petri nets. Pioneering papers on data flow were based on the firing rule semantics of Petri nets [Den80]. On the other hand the development of data flow influenced the generalization of Petri nets. High-level Petri nets are special cases of such data flow diagrams. Both places and transitions in Petri nets can be seen as data flow nodes.

Functional system specification. In this section we give a brief summary of the basic mathematical concepts of stream-based functional system models. We consider system components with a finite number of input and output channels. Messages are exchanged over the channels. A channel history is mathematically modeled by a stream of messages. The behavior of a (deterministic) component corresponds to a function mapping the streams on its input channels onto streams for its output channels.

A *stream* of messages over a given message set M is a finite or infinite sequence of messages. We define the set of stream by

$$M^\omega =_{\text{def}} M^* \cup M^\infty.$$

By $x \widehat{\ } y$ we denote the result of concatenating two streams x and y . We assume that $x \widehat{\ } y = x$, if x is infinite. By $\langle \rangle$ we denote the empty stream. For simplicity we write for $a \in M$, $x \in M^\omega$

$$a \widehat{\ } x \text{ instead of } \langle a \rangle \widehat{\ } x \quad \text{and} \quad x \widehat{\ } a \text{ instead of } x \widehat{\ } \langle a \rangle.$$

If a stream x is a *prefix* of a stream y , we write $x \sqsubseteq y$. The relation \sqsubseteq is called *prefix order*. It is formally specified by

$$x \sqsubseteq y =_{\text{def}} \exists z \in M^\omega : x \widehat{\ } z = y.$$

The relation \sqsubseteq is a partial order on the set of streams. The empty stream $\langle \rangle$ is the least element.

Given a partially ordered set, a subset is called *directed* if, for any pair of elements in S , there exists an upper bound in S . A partially ordered set is called *complete* if, for every directed set of streams, there exists a least upper bound. The set of streams ordered by the prefix order is complete. The least upper bound of a directed set S is denoted by *lub* S .

The behavior of deterministic interactive systems with n input channels and m output channels is modeled by prefix monotonic functions

$$f : (M^\omega)^n \rightarrow (M^\omega)^m$$

called (m, n) -*ary stream processing functions*.

A function f mapping a complete partially ordered set onto a complete partially ordered set is called *continuous*, if, for every directed set S ,

$$f(\text{lub } S) = \text{lub } \{f(x) \mid x \in S\}$$

The set of all prefix continuous stream processing functions of functionality $(M^\omega)^n \rightarrow (M^\omega)^m$ is denoted by

$$SPF_m^n.$$

For simplicity, we do not consider type information here and assume M to be just a set of messages.

The following functions on streams are useful in specifications:

$$\begin{array}{ll} rt : M^\omega \rightarrow M^\omega & \text{rest of a stream} \\ ft : M^\omega \rightarrow M \cup \{\perp\} & \text{first element of a stream} \\ \# : M^\omega \rightarrow \mathbb{N} \cup \{\infty\} & \text{length of a stream} \\ \text{-}\odot\text{-} : \mathcal{P}(M) \times M^\omega \rightarrow M^\omega & \text{filter of a stream} \end{array}$$

Here \perp is used as a dummy to avoid partial functions. These functions are easily specified by the following algebraic equations (let $x \in M^\omega$, $m \in M$, $S \subseteq M$):

$$\begin{aligned} rt(\langle \rangle) &= \langle \rangle, & rt(m \widehat{\ } x) &= x, \\ ft(\langle \rangle) &= \perp, & ft(m \widehat{\ } x) &= m, \\ \#(\langle \rangle) &= 0, & \#(m \widehat{\ } x) &= 1 + \#(x), \\ S \odot \langle \rangle &= \langle \rangle, \\ S \odot (m \widehat{\ } x) &= m \widehat{\ } (S \odot x), \text{ if } m \in S \\ S \odot (m \widehat{\ } x) &= S \odot x, \text{ if } m \notin S \end{aligned}$$

These axioms specify the functions completely. They are useful in proofs, too.

Stream processing functions can easily be specified by logical formulas in the style of algebraic equations as is demonstrated for the running example below. Given such functions, we may compose them.

We use two forms of composition: parallel composition and sequential composition. Given functions $f \in SPF_k^n$, $g \in SPF_m^k$ we write

$$f;g$$

for the *sequential composition* of f and g which yields a function in SPF_m^n , where

$$(f;g)(x) = g(f(x)).$$

Given functions $f \in SPF_m^n$, $g \in SPF_{m'}^{n'}$ we write

$$f||g$$

for the *parallel composition* of f and g which yields a function in $SPF_{m+m'}^{n+n'}$, where (let $x \in (M^\omega)^n$, $y \in (M^\omega)^{n'}$):

$$(f||g)(\langle x, y \rangle) = \langle f(x), g(y) \rangle$$

Finally, given a function

$$f \in SPF_m^n$$

we may construct a function by the feedback operator leading an output line back to an input line. We write

$$\mu_j^k f \in SPF_{m-1}^{n-1}$$

for the function defined by the equation ($1 \leq k \leq n$, $1 \leq j \leq m$)

$$\mu_j^k f(x_1, \dots, x_{k-1}, x_{k+1}, \dots, x_n) = (y_1, \dots, y_{j-1}, y_{j+1}, \dots, y_m)$$

where z is the prefix least stream such that the following equation holds

$$f(x_1, \dots, x_{k-1}, z, x_{k+1}, \dots, x_n) = (y_1, \dots, y_{j-1}, z, y_{j+1}, \dots, y_m)$$

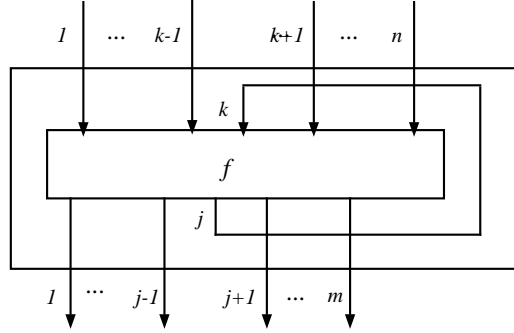


Fig. 13.6. Data flow graph for feedback

See Figure 13.6.

Since f is prefix monotonic, such a stream least solution (least fix point) always exists. Of course, it is unique.

By $SPEC_m^n$ we denote the set of all predicates Q where

$$Q: SPF_m^n \rightarrow \{T, F\}$$

The set $SPEC_m^n$ denotes the set of all component specifications for a component with n input channels and m output channels.

We want to compose specifications of components to networks. Each form of composition introduced for functions can be extended to component specifications in a straightforward way. Given component specifications $Q \in SPEC_k^n, R \in SPEC_m^k$ we write

$$Q; R$$

for the predicate in $SPEC_m^n$ where

$$(Q; R)(f) \Leftrightarrow \exists q, r. f = q; r \wedge Q(q) \wedge R(r)$$

Trivially, we have for all specifications $Q \in SPEC_m^n$ the following equations, where I denotes the identity function:

$$Q; I = Q \quad \text{and} \quad I; Q = Q.$$

Given specifications $Q \in SPEC_m^n, R \in SPEC_{m'}^{n'}$ we write

$$Q \parallel R$$

for the predicate in $SPEC_{m+m'}^{n+n'}$ where

$$(Q \parallel R)(f) \Leftrightarrow \exists q, r. f = q \parallel r \wedge Q(q) \wedge R(r).$$

Feedback also carries over in a straightforward manner to specifications.

$$(\mu_j^k Q)(f) \equiv \exists f'. Q(f') \wedge f = (\mu_j^k f')$$

Any data flow graph can be described by parallel composition and feedback. This is easily seen. To build a compositional form for a given data flow diagram where the nodes are described by specifications of stream processing functions, we form a large parallel composition of all data flow nodes. Then we connect the output lines to input lines by feedback as shown by the data flow diagram.

A specification $\overline{Q} \in SPEC_m^n$ is called a *property refinement* of a specification $Q \in SPEC_m^n$ if, for all functions f , we have $\overline{Q}(f) \Rightarrow Q(f)$. We write then

$$\overline{Q} \Rightarrow Q$$

In other words, \overline{Q} is a property refinement of Q if the set of functions described by \overline{Q} is a subset of the set of functions described by Q . More sophisticated notions of refinement are obtained by abstraction and representation specifications as introduced in [Bro97].

A pair of specifications A and R are called abstraction and representation, if

$$R; A = I$$

Let A_1 be an abstraction specification and R_2 be a representation specification. A specification C' is called a refinement of specification C if we have

$$C' \Rightarrow A_1; C; R_2$$

Given the corresponding abstraction specification A_2 and a representation specification R_1 , the identities

$$R_1; A_1 = I \qquad R_2; A_2 = I$$

allows us to deduce

$$R_1; C'; A_2 \Rightarrow C.$$

The actual specification of data flow nodes can be done by logical formulas describing the relationship between the input and output streams.

The strong aspect of stream processing concepts is their modularity. They allow for a modular specification, composition, and refinement of interacting systems.

Example 13.7. (Bit using stream functions)

First we specify the two components, buffer and user.

For each component we first give its functionality, then we give the specifying axioms.

$$BUFFER: (\cup \{ GET \})^\omega \rightarrow \omega$$

$$\begin{aligned} \forall h \in \Sigma^*, z \in \Sigma^*, x \in (\Sigma \cup \{GET\})^\omega: \\ BUFFER(h \hat{\ } z \hat{\ } GET \hat{\ } x) &= h \hat{\ } BUFFER(z \hat{\ } x) \vee \\ BUFFER(h \hat{\ } z \hat{\ } GET \hat{\ } x) &= h \hat{\ } BUFFER(h \hat{\ } z \hat{\ } GET \hat{\ } x) \vee \\ BUFFER(h \hat{\ } z \hat{\ } GET \hat{\ } x) &= \langle 10^{10} \rangle \end{aligned}$$

The user is modeled by the function

$$USER: \Sigma^\omega \times \{START\}^\omega \rightarrow (\Sigma \cup \{GET\})^\omega \times \{OK, ERROR\}^\omega$$

which is specified by the equation

$$USER(x, \langle START \rangle) = \langle 0 \hat{\ } 1 \hat{\ } GET, INSPECT(x) \rangle$$

where the auxiliary function

$$INSPECT: \Sigma^\omega \rightarrow \{OK, ERROR\}^\omega$$

is specified by the equations ($i \in \Sigma \wedge x \in \Sigma^*$)

$$\begin{aligned} INSPECT(0 \hat{\ } x) &= \langle OK \rangle \\ INSPECT(i \hat{\ } x) &= \langle ERROR \rangle, \quad i \neq 0 \end{aligned}$$

The system is formed by the parallel composition of *BUFFER* and *USER*, and a feedback of *Output_I* to *Input* and of *Output* to *Input_I*; see its graphical explanation in Figure 13.7.

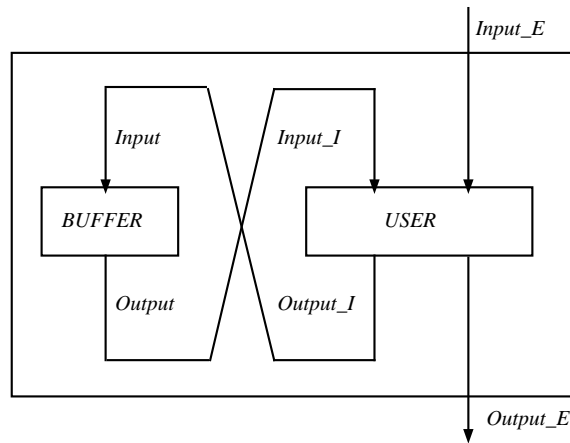


Fig. 13.7. Structure of **Bit** specified using stream-processing functions

The structure of the data flow diagram is captured logically by the equations

$$\begin{aligned} Output &= BUFFER(Input) \\ (Output_I, Output_E) &= USER(Input_I, Input_E) \\ Input_I &= Output \\ Input &= Output_I \\ Input_E &= \langle START \rangle \end{aligned}$$

Given these equations and the specifying equations above, we may begin a straightforward reasoning about the value of $Output_E$.

Integration with algebraic data type specifications. The integration of stream processing functions with algebraic specification is quite simple. Streams are nothing other than an abstract data type, just slightly more complex because they may be infinite. The data types forming the messages in a stream are easily specified by algebraic specifications.

Stream processing functions can also be specified by algebraic specifications. Similarly sequential composition, parallel composition, and feedback can be described by algebraic specifications. Here we need higher-order algebraic specifications, of course. The algebraic equations for the composition operators lead to a rich algebra of stream processing components.

13.3 Dynamic-data types (A3)

As shown in the section on CCS, labeled transition systems (lts) are an effective way to give an operational semantics to a process algebra. In this section we show how lts can be directly used for the specification of system behavior.

13.3.1 Labelled transition logic (LTL)

The main references for LTL are [AR87,AR96b,CR97]; the first appearance is in [AMRW85]. Notice that in the past the terms “algebraic transition systems” (e.g., in [AMRW85,AR87]) and “dynamic specifications” (e.g., in [CR97,Reg93,AR96a]) have also been used for the specifications built using LTL.

To model processes LTL uses labeled transition systems, see Section 13.1.1, and supplies two different kinds of specifications at different levels of abstraction:

- requirement** for expressing the requirements on a concurrent system; a requirement specification should determine a class of nonisomorphic models, all those formally and abstractly describing systems having such requirements;
- design** for expressing the abstract design of a concurrent system, to define abstractly and formally the way in which we intend to design the system; a design specification should determine one model, the one formally and abstractly describing the designed system.

LT-Structures. An lts can be represented by a first-order structure (an algebra with predicates) \mathbf{A} on a signature with at least two sorts, *state* and *label*, whose elements correspond respectively to the states and labels of the system, and a predicate $_ \xrightarrow{_} _ : \text{state} \times \text{label} \times \text{state}$ representing the transition relation. The triple $\langle |A|_{\text{state}}, |A|_{\text{label}}, \rightarrow_{\mathbf{A}} \rangle$ is the corresponding lts. Obviously we can have lts whose states are built by states of other lts (for modeling concurrent systems); in such a case we use structures whose signature has different sorts corresponding to states and labels, and different predicates corresponding to transition relations.

In a formal model for concurrent systems we may need to consider data, too (for example, the data manipulated by a system such as natural numbers); to handle these cases we consider structures which also have sorts that just correspond to data and not to states or labels of lts.

The first-order structures (algebras) corresponding to lts are called *LT-structures* and are formally defined as follows.

- An *LT-signature* $LT\Sigma$ is a pair $\langle \Sigma, DS \rangle$, where:
 - * $\Sigma = \langle S, \Omega, \Pi \rangle$ is a first-order signature,
 - * $DS \subseteq S$ (the elements in DS are the *dynamic sorts*, which are the sorts corresponding to states of lts),
 - * for all $ds \in DS$ there exist a sort $lab_ds \in S - DS$ (labels of the transitions of the processes of sort ds) and a predicate $_ \xrightarrow{_} _ : ds \times lab_ds \times ds \in \Pi^3$ (transition relation of the dynamic elements of sort ds).
- An *LT-structure* on $LT\Sigma$ (abbreviated to $LT\Sigma$ -structure) is a Σ -first-order structure (a Σ -algebra with predicates).

Design LT-specifications. LT-specifications are particular algebraic/logic specifications for LT-structures where conditional formulas are used as axioms; since transitions are described by predicate symbols, such formulas also allow us to express properties on the activity of processes.

An *LT-specification* is a pair $SP = \langle LT\Sigma, AX \rangle$, where $LT\Sigma = \langle \Sigma, DS \rangle$ is an LT-signature and AX a set of conditional formulas on $LT\Sigma$ having form

$$\bigwedge_{i=1, \dots, n} \alpha_i \Rightarrow \alpha_{n+1},$$

where for $i = 1, \dots, n + 1$, α_i is an *atom*, i.e., a formula having the form either $t = t'$ or $p(t_1, \dots, t_m)$.

We can give SP different semantics, as initial and observational, briefly presented below.

The initial semantics of SP determines one (up to isomorphism) LT-structure, precisely $IMod(SP) = T_{\Sigma} / \equiv_{AX}$, where \equiv_{AX} is the congruence

³ In this paper, for some of the operation and predicate symbols we use a mixfix notation; for instance, $_ \xrightarrow{_} _ : ds \ lab_ds \ ds$ means that we shall write $t \xrightarrow{t'} t''$ instead of $\rightarrow (t, t', t'')$.

over T_{Σ} generated by the Birkhoff deductive system for conditional specifications, sound and complete w.r.t. the models of SP and the atomic formulas, see Chapter 3.

Since in an LT-structure the transitions correspond to the truth of the ground atoms built by the transition predicates, we have that the transitions in the initial model of an LT-specification are just those whose corresponding atoms can be proved by using the Birkhoff system.

In most cases the initial semantics of an LT-specification is too fine, since it takes into account all details of the activity of the processes (as intermediate states). It may happen that we want to consider as semantically equivalent two processes having different associated transition trees, see Section 13.1.1.

There is a general way to give an observational semantics to LT-specifications introduced in the general case of conditional specifications by some of the authors and A. Giovini (see [AGR92] for a full presentation); this approach is well suited for use in LT-specifications, which specify concurrent systems, since it generalizes and extends the Milner–Park’s bisimulation technique to a purely algebraic setting.

Example 13.8. (Bit using LTL) We first specify the two components of the system, the buffer and the user, and then how they cooperate.

Below “**dsort** . . . : . . .” is the construct for declaring dynamic sorts, the second argument is the syntactic form of the transition predicate; thus

dsort *buffer* : $_ \xrightarrow{_} _$

declares the dynamic sort *buffer*, the associated sort of the labels *lab_buffer*, and the transition predicate $_ \xrightarrow{_} _ : \text{buffer} \times \text{lab_buffer} \times \text{buffer}$.

```

spec BUFFER =
  enrich INT_QUEUE[buffer/queue] by
  dsorts  buffer :  $\_ \xrightarrow{\_} \_$ 
  opns     $\tau : \rightarrow \text{lab\_buffer}$ 
          RECEIVE, RETURN :  $\text{int} \rightarrow \text{lab\_buffer}$ 
  axioms  Not_Empty(b)  $\Rightarrow b \xrightarrow{\text{RETURN}(\text{First}(b))} \text{Remove}(b)$ 
           $b \xrightarrow{\text{RECEIVE}(i)} \text{Put}(i, b)$ 
           $b \xrightarrow{\tau} \text{Put}(10^{10}, \text{Empty})$ 
          Not_Empty(b)  $\Rightarrow b \xrightarrow{\tau} \text{Dup}(b)$ 

spec USER_STATUS =
  enrich INT by
  sorts   user_status
  opns    Initial, Putting_0, Putting_1, Reading, Terminated :  $\rightarrow \text{user\_status}$ 
          Read :  $\text{int} \rightarrow \text{user\_status}$ 

spec USER =
  enrich USER_STATUS[user/user_status] by
  dsorts  user :  $\_ \xrightarrow{\_} \_$ 
  opns    START, OK, ERROR :  $\rightarrow \text{lab\_user}$ 
          PUT, GET :  $\text{int} \rightarrow \text{lab\_user}$ 

```

axioms $Initial \xrightarrow{START} Putting_0$
 $Putting_0 \xrightarrow{PUT(0)} Putting_1$
 $Putting_1 \xrightarrow{PUT(1)} Reading$
 $Reading \xrightarrow{GET(i)} Read(i)$
 $Read(0) \xrightarrow{OK} Terminated$
 $NotEq(i, 0) \Rightarrow Read(i) \xrightarrow{ERROR} Terminated$

spec $SYSTEM =$

enrich $BUFFER + USER$ by

dsorts $system: - \xrightarrow{-} -$

opns $- | -: buffer \times user \rightarrow system$

$START, OK, ERROR, \tau: \rightarrow lab_system$

axioms $u \xrightarrow{START} u' \Rightarrow Empty | u \xrightarrow{START} Empty | u'$
 $b \xrightarrow{RECEIVE(i)} b' \wedge u \xrightarrow{PUT(i)} u' \Rightarrow b | u \xrightarrow{\tau} b' | u'$
 $b \xrightarrow{RETURN(i)} b' \wedge u \xrightarrow{GET(i)} u' \Rightarrow b | u \xrightarrow{\tau} b' | u'$
 $u \xrightarrow{OK} u' \Rightarrow b | u \xrightarrow{OK} b | u'$
 $u \xrightarrow{ERROR} u' \Rightarrow b | u \xrightarrow{ERROR} b | u'$
 $b \xrightarrow{\tau} b' \Rightarrow b | u \xrightarrow{\tau} b' | u$

Requirement LT-specifications. $SP = \langle LT\Sigma, AX \rangle$ with loose semantics determines the class of its *models*, $Mod(SP)$, i.e., all $LT\Sigma$ -structures satisfying all formulas in AX .

LT-specifications with loose semantics can be used to specify the requirements on a concurrent system, thus determining a class of systems (all those satisfying such requirements), instead of abstractly defining one particular system. However, conditional formulas are too limited to express all relevant requirements on concurrent systems, thus various extensions of first-order logic are used, e.g., including combinators of the branching-time temporal logic [CR97], the deontic logic [CR96], using the concept of “abstract event” [AR93], etc. Below we briefly present the extension of [CR97] with branching-time temporal combinators (see Section 13.2.3).

Let $LT\Sigma = \langle \langle S, \Omega, \Pi \rangle, DS \rangle$ be an LT-signature, L an $LT\Sigma$ -structure, and $ds \in DS$. We need the following technical definitions. $PATH(L, ds)$ denotes the set of the *paths* for the elements of sort ds , i.e., all sequences of transitions having the form either (1) or (2) below:

- (1) $d_0 l_0 d_1 l_1 d_2 l_2 \dots$
- (2) $d_0 l_0 d_1 l_1 d_2 l_2 \dots d_n \quad n \geq 0$

where for all $i \geq 0$, $d_i \in |L|_{ds}$, $l_i \in |L|_{lab_ds}$, and $(d_i, l_i, d_{i+1}) \in \rightarrow_L$.

$FirstS(\sigma)$ denotes the *first state* of σ ; and $FirstL(\sigma)$ denotes the *first label* of σ , if exists, i.e., if σ is not just a state.

$\sigma \in PATH(L, ds)$ is *maximal* iff either it is infinite or there do not exist l, d' such that $\langle d_n, l, d' \rangle \in \rightarrow_L$.

Given $\sigma = d_0 l_0 d_1 l_1 d_2 l_2 \dots$ and $h \geq 0$, if d_h exists, then $\sigma|_h$ denotes the path $d_h l_h d_{h+1} l_{h+1} d_{h+2} \dots$, otherwise undefined.

The set of *formulas*, denoted by $F(LT\Sigma, X)$, and the family of the sets of *path formulas*, denoted by $\langle PF(LT\Sigma, X)_{ds} \rangle_{ds \in DS}$, on $LT\Sigma$, and variables X are defined by multiple induction as follows. For each $s \in S$ and $ds \in DS$:

formulas

- $p(t_1, \dots, t_n) \in F(LT\Sigma, X)$ $p: s_1 \times \dots \times s_n \in \Pi, t_i \in |T_{LT\Sigma}(X)|_{s_i}$
- $t_1 = t_2 \in F(LT\Sigma, X)$ $t_1, t_2 \in |T_{LT\Sigma}(X)|_s$
- $\neg \phi, \phi \Rightarrow \phi', \forall x. \phi \in F(LT\Sigma, X)$ $\phi, \phi' \in F(LT\Sigma, X), x \in X$
- $\Delta(t, \pi) \in F(LT\Sigma, X)$ $t \in |T_{LT\Sigma}(X)|_{ds}, \pi \in PF(LT\Sigma, X)_{ds}$

path formulas

- $[\lambda x. \phi] \in PF(LT\Sigma, X)_{ds}$ $x \in X_{ds}, \phi \in F(LT\Sigma, X)$
- $\langle \lambda x. \phi \rangle \in PF(LT\Sigma, X)_{ds}$ $x \in X_{lab_ds}, \phi \in F(LT\Sigma, X)$
- $\pi_1 \mathcal{U} \pi_2 \in PF(LT\Sigma, X)_{ds}$ $\pi_1, \pi_2 \in PF(LT\Sigma, X)_{ds}$
- $\neg \pi, \pi \Rightarrow \pi', \forall x. \pi \in PF(LT\Sigma, X)_{ds}$ $\pi, \pi' \in PF(LT\Sigma, X)_{ds}, x \in X_s$

The formulas of such logic include the usual formulas of first-order logic with equality; if $LT\Sigma$ contains dynamic sorts, they also include formulas built with the transition predicates.

The formula $\Delta(t, \pi)$ can be read as “for every path σ starting in the state denoted by t , the path formula π holds on σ ”. We anchor these formulas to states, following the ideas in [MP89]. The major difference with the classical temporal logic of Section 13.2.3 is that we do not specify a single system but, in general, one or many types of systems, so there is not a single initial state but several, hence the need for an explicit reference to states (through terms) in the formulas built with Δ . The formula $[\lambda x. \phi]$ holds on the path σ whenever ϕ holds at the first state of σ ; while the formula $\langle \lambda x. \phi \rangle$ holds on the path σ if σ is not just a single state and ϕ holds at the first label of σ .

Let L be an $LT\Sigma$ -structure and v a variable evaluation of X in L ; then we define by multiple induction:

- the validity of $\phi \in F(LT\Sigma, X)$ in L w.r.t. v (written $L, v \models \phi$),
- the validity of $\pi \in PF(LT\Sigma, X)$ on a path σ in L w.r.t. v (written $L, v, \sigma \models \pi$),

as follows:

- $L, v \models p(t_1, \dots, t_n)$ iff $\langle v^\#(t_1), \dots, v^\#(t_n) \rangle \in p_L$
- $L, v \models t_1 = t_2$ iff $v^\#(t_1) = v^\#(t_2)$
- $L, v \models \Delta(t, \pi)$ iff for each $\sigma \in PATH(L, ds)$ such that $FirstS(\sigma) = v^\#(t)$, $L, v, \sigma \models \pi$
- $L, v, \sigma \models [\lambda x. \phi]$ iff $L, v[FirstS(\sigma)/x] \models \phi$
- $L, v, \sigma \models \langle \lambda x. \phi \rangle$ iff $FirstL(\sigma)$ is defined and $L, v[FirstL(\sigma)/x] \models \phi$
- $L, v, \sigma \models \pi_1 \mathcal{U} \pi_2$ iff there exists $j \geq 0$ such that for all $h, 0 < h < j$, $L, v, \sigma|_h \models \pi_1$ and $L, v, \sigma|_j \models \pi_2$
- $\neg \phi, \phi \Rightarrow \phi', \forall x. \phi, \neg \pi, \pi \Rightarrow \pi', \forall x. \pi$ as usual.

ϕ is *valid* in L (written $L \models \phi$) iff $L, v \models \phi$ for all evaluations v .

In the above definitions we have used a minimal set of combinators. However it is possible to define other, derived, combinators as for the classical logics of Section 13.2.3; plus $\nabla(t, \pi) =_{\text{def}} \neg \Delta(t, \neg \pi)$ (which means at least in one case, i.e., the property represented by π holds at least on one path).

Example 13.9. (Bit using LTL (requirements)) As already done in Example 13.6, we give here some sample requirements on **Bit**; but in a different way to before, we specify the system modularly, by considering its components first and then how they are put together in order to cooperate. Furthermore, temporal LTL has also edge formulas, so we can also conveniently express properties concerning the interactions of the system with the environment. Since properties are anchored to processes (concurrent systems), we can relate properties of the system to properties of its components.

Below $\langle \lambda l. l = t \rangle$ is abbreviated to $\langle t \rangle$.

```
spec USER =
  enrich INT by
  dsorts  user:  $\_ \xrightarrow{\_} \_$ 
  preds   Terminated: user
  opns    PUT, GET: int  $\rightarrow$  lab_user
  axioms  Terminated(u)  $\Rightarrow$ 
           $\Delta(u, [\lambda u'. \text{Terminated}(u')])$ 
          ** if the user is terminated, it remains so forever (R1)
```

```
spec BUFFER =
  enrich INT by
  dsorts  buffer:  $\_ \xrightarrow{\_} \_$ 
  opns    RECEIVE, RETURN: int  $\rightarrow$  lab_buffer
  axioms  b  $\xrightarrow{\text{RECEIVE}(i)}$  b'  $\Rightarrow \Delta(b', [\lambda x. \exists x'. x \xrightarrow{\text{RETURN}(i)} x'])$ 
          ** after receiving  $i$  the buffer eventually will have
          ** the capability to return  $i$ 
           $\nabla(b, \langle \text{RECEIVE}(i) \rangle)$ 
          ** the buffer must be able to receive any integer (R0)
```

```
spec SYSTEM =
  enrich BUFFER + USER by
  dsorts  system:  $\_ \xrightarrow{\_} \_$ 
  opns    START, OK, ERROR,  $\tau$ :  $\rightarrow$  lab_system
           $\_ \mid \_$ : buffer  $\times$  user  $\rightarrow$  system
  axioms   $\exists b, u. s = b \mid u$ 
           $\langle b, u \rangle \xrightarrow{l} \langle b', u' \rangle \Rightarrow$ 
             $(b = b' \wedge u \xrightarrow{l} u') \vee$ 
             $(u = u' \wedge b \xrightarrow{l} b') \vee$ 
             $(\exists i. l = \tau \wedge u \xrightarrow{\text{GET}(i)} u' \wedge b \xrightarrow{\text{RETURN}(i)} b') \vee$ 
             $(\exists i. l = \tau \wedge u \xrightarrow{\text{PUT}(i)} u' \wedge b \xrightarrow{\text{RECEIVE}(i)} b')$ 
          ** (R5)
```

$$\nabla(s, \langle \lambda l. \neg l = ERROR \rangle)$$

** there always exists a possible “correct” behavior (**R2**)

$$Terminated(u) \Rightarrow$$

$$\Delta(b \mid u, [\lambda s. \exists b', u'. (s = b' \mid u' \wedge Terminated(u'))] \mathcal{U}$$

$$[\lambda s. \neg \exists l, s'. s \xrightarrow{l} s'])$$

** if the user is terminated, it remains so until the system stops

$$s \xrightarrow{START} s' \Rightarrow \Delta(s', (\langle OK \rangle \vee \langle ERROR \rangle))$$

** after the system has been started, in any case eventually

** it will send out either *OK* or *ERROR* (**R3**)

$$(s \xrightarrow{OK} s' \vee s \xrightarrow{ERROR} s') \Rightarrow$$

$$\Delta(s', \langle \lambda l. \neg (l = OK \vee l = ERROR) \rangle)$$

** *OK* and *ERROR* are sent at most once, and it cannot

** happen that both are sent (**R4**)

$$\nabla(b \mid u, \langle \tau \rangle) \Rightarrow \nabla(b, \langle \tau \rangle)$$

** if the system may eventually only do internal actions,

** then the buffer component has such a possibility, too

Some of the axioms of the above specifications are just to show the peculiarity of this logic. For example, the unique axiom of *USER* requires that it must remain terminated in isolation; while the axiom of *SYSTEM* about terminated requires something about the behavior of the user when put within the system. The last axiom of *SYSTEM* shows how properties of the components can be related to properties of the whole system.

13.3.2 Rewriting logic (RL)

RL is a formalism for the specification of concurrent systems developed by Meseguer in the recent years, sharing some of the ideas of LTL; moreover, its specifications are syntactically very similar to those of LTL. For both formalisms the behavior of processes is modeled by means of transition systems; the states of such systems are elements of some carriers of an algebra, given as the initial model of a conditional specification; the structure of a term representing one of such states models the concurrent structure of the system in that state; and the transitions are defined by conditional formulas in which the transition symbol (arrow) appears.

Clearly, there are also major differences between the two formalisms: the transitions are labeled in the case of LTL and nonlabeled for RL, and have special properties in the RL case, as to be closed by reflexivity, transitivity, and congruence w.r.t. the operations; and, most important, their intuitive interpretation is very different in the two cases:

LTL $t \xrightarrow{l} t'$ means that the system in the state represented by t has the “capability” of passing into the state represented by t' by performing some “atomic” activity, i.e., an activity that cannot be interrupted, where no information on the intermediate states is available, whose interaction with the environment is represented by l , and at each instant a system can perform only one of these activities.

RL $t \longrightarrow t'$ means that the system in the state represented by t can pass into the state represented by t' by performing some activity completely independently from the environment; such activity may be also the composition of several “smaller” activities of the same system, and so information on the intermediate states may be available (for a terminating system, for example, we may have transitions which correspond to whole evolutions of the system from the beginning till the termination).

A complete study of the relationship between **RL** and **LTL** can be found in [AR97b].

Below we give a short presentation of **RL**, the main references are [Mes92, MM93]; notice that in such papers Meseguer has used the language of category theory to present **RL**, while here, for clarity, we use a more logic-algebraic style.

A *rewrite theory*, i.e., an **RL** specification, is a 4-tuple $\mathcal{R} = \langle \Sigma, E, L, R \rangle$, where $\Sigma = \langle S, \Omega \rangle$ is a signature, E a set of equations on Σ , and R a set of *rewrite rules* of the form

$$r: [t] \longrightarrow [t'] \text{ if } [u_1] \longrightarrow [v_1] \wedge \dots \wedge [u_k] \longrightarrow [v_k]$$

with $r \in L$ and $[t], [t'], [u_1], [v_1], \dots, [u_k], [v_k] \in T_\Sigma(X)/\equiv_E$.

The *entailment system* associated with \mathcal{R} has the following rules:

1. Reflexivity For each $[t] \in T_\Sigma(X)/\equiv_E$
$$\frac{}{[t] \longrightarrow [t]}$$

2. Congruence For each $f: s_1 \times \dots \times s_n \rightarrow s \in \Omega$

$$\frac{\frac{[t_1] \longrightarrow [t'_1] \dots [t_n] \longrightarrow [t'_n]}{[f(t_1, \dots, t_n)] \longrightarrow [f(t'_1, \dots, t'_n)]}}{[f(t_1, \dots, t_n)] \longrightarrow [f(t'_1, \dots, t'_n)]}}$$

3. Replacement' For each rewrite rule

$$r: [t(\bar{x})] \longrightarrow [t'(\bar{x})] \text{ if } [u_1(\bar{x})] \longrightarrow [v_1(\bar{x})] \wedge \dots \wedge [u_k(\bar{x})] \longrightarrow [v_k(\bar{x})]$$

belonging to R , where \bar{x} is the vector of all variables appearing in the rule and \bar{w} a corresponding vector of elements in $T_\Sigma(X)/\equiv_E$

$$\frac{[u_1(\bar{w}/\bar{x})] \longrightarrow [v_1(\bar{w}/\bar{x})] \dots [u_k(\bar{w}/\bar{x})] \longrightarrow [v_k(\bar{w}/\bar{x})]}{[t(\bar{w}/\bar{x})] \longrightarrow [t'(\bar{w}/\bar{x})]}$$

4. Transitivity
$$\frac{[t_1] \longrightarrow [t_2] \quad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$$

Here, for simplicity, we use the entailment system above, which is a slightly modified version of the original [Mes92]: rule 3, shown below, has been changed to avoid the simultaneous rewriting of an element substituted for a variable. In [AR97b] we show that this entailment system is equivalent

to the original, since the entailed sequents are the same and the structure of the proofs is preserved.

Now we define the models of the rewrite theories using the entailment system.

- An \mathcal{R} -presystem is a direct reflexive graph $G = (\theta_0, \theta_1 : Edges \rightarrow Nodes, id)$, where $id : Nodes \rightarrow Edges$ such that $\theta_0(id(n)) = \theta_1(id(n)) = n$ (id expresses the reflexivity of the graph) together with:
 - a Σ -structure on $Nodes$ such that it satisfies E and the edges respect the sorts (i.e., for each edge e , $\theta_0(e)$ and $\theta_1(e)$ have the same sort);
 - for each $f : s_1 \times \dots \times s_n \rightarrow s \in \Omega$, a partial operation $f : Edges^n \dashrightarrow Edges$ such that $f(e_1, \dots, e_n)$ is defined iff for $i = 1, \dots, n$, e_1, \dots, e_n are edges of sorts s_1, \dots, s_n respectively, and

$$\theta_0(f(e_1, \dots, e_n)) = f(\theta_0(e_1), \dots, \theta_0(e_n)),$$

$$\theta_1(f(e_1, \dots, e_n)) = f(\theta_1(e_1), \dots, \theta_1(e_n));$$
 - a partial operation $_ ; _ : Edges^2 \dashrightarrow Edges$ such that $e; e'$ is defined iff e and e' have the same sort and $\theta_1(e) = \theta_0(e')$ and $\theta_0(e; e') = \theta_0(e)$, $\theta_1(e; e') = \theta_1(e')$;
 - for each rewrite rule $r : [t(\bar{x})] \longrightarrow [t'(\bar{x})]$ if $[u_1(\bar{x})] \longrightarrow [v_1(\bar{x})] \wedge \dots \wedge [u_k(\bar{x})] \longrightarrow [v_k(\bar{x})]$ belonging to R , a partial operation⁴ $r : VarEv \times Edges^k \dashrightarrow Edges$, where $VarEv$ is the set of variable evaluations from \bar{x} into $Nodes$, such that $r(v, e_1, \dots, e_k)$ is defined iff, for $i = 1, \dots, k$, $\theta_0(e_i) = v^\#(u_i(\bar{x}))$ and $\theta_1(e_i) = v^\#(v_i(\bar{x}))$, and $\theta_0(r(v, e_1, \dots, e_k)) = v^\#(t(\bar{x}))$, $\theta_1(r(v, e_1, \dots, e_k)) = v^\#(t'(\bar{x}))$.
- A *morphism* ϕ between two \mathcal{R} -presystems P and P' is a graph morphism which preserves the Σ -structure on the nodes and the operations on the edges.
- An \mathcal{R} -system is an \mathcal{R} -presystem satisfying the following equations on edges (the adaptation of those of [Mes92] to our modified entailment system):
 1. **Category** $(e; e'); e'' = e; (e'; e'')$
 2. **Functoriality of the Σ -structure**
for each $f : s_1 \times \dots \times s_m \rightarrow s \in \Omega$

$$f(e_1; e'_1, \dots, e_m; e'_m) = f(e_1, \dots, e_m); f(e'_1, \dots, e'_m)$$

$$f(id(n_1), \dots, id(n_m)) = id(f(n_1, \dots, n_m))$$
 3. **Axioms in E** For each $t(x_1, \dots, x_n) = t'(x_1, \dots, x_n) \in E$

$$t(e_1, \dots, e_n) = t'(e_1, \dots, e_n).$$
- $\mathcal{T}_{\mathcal{R}}$ is the initial element in the class of the \mathcal{R} -systems, i.e., the \mathcal{R} -system where $Nodes$ is T_{Σ}/\equiv_E , the edges are generated by the operations with edge types, and the edges represented by two different terms are identified iff their identification follows from equations 1, 2, and 3 above.

⁴ If several rules with the same label will result to operations of the same functionality, then we assume that the names of such operations are made different.

The ground terms built by the edge operations bijectively correspond to the *proofs* of sequents in the entailment systems associated with \mathcal{R} ; and the axioms on edges correspond to the required identifications on the proofs.

\mathcal{R} -systems, and therefore also $\mathcal{T}_{\mathcal{R}}$, can be seen as categories, where the objects are the nodes, the morphisms are the edges, and “;” is the composition operation on morphisms; axiom 1 and the fact that the graph is reflexive ensure that they really are categories.

$\mathcal{T}_{\mathcal{R}}$ is considered the standard semantics of a theory \mathcal{R} .

We can see a striking difference between **RL** and **LTL**: for **LTL** the precise form and number of the axioms of a specification is irrelevant, while the precise form and number of rules of a theory is extremely important for **RL**. If two **RL** theories \mathcal{R} and \mathcal{R}' have the same signature, equivalent sets of equations, and $\rightarrow_{\mathcal{R}} = \rightarrow_{\mathcal{R}'}$, then in general $\mathcal{T}_{\mathcal{R}}$ and $\mathcal{T}_{\mathcal{R}'}$ are not isomorphic.

In **LTL** we have that for each specification there exist infinite isomorphic specifications with different sets of axioms (e.g., they can be obtained by adding derived axioms).

Furthermore, the rule labels are not relevant in **RL**, only the rules are; indeed the same label can be used for several rules but the edges and the operations over them are determined by the rules, not by the labels; and we have that applications of operations associated with different rules, labeled in the same way with premises and consequences of the same sorts, must be disambiguated.

Example 13.10. (Bit using RL)

```
spec BUFFER =
  enrich INT_QUEUE[buffer/queue] by
  rl    $\tau_1$ :     $b \longrightarrow Dup(b)$ 
  rl    $\tau_2$ :     $b \longrightarrow Put(10^{10}, Empty)$ 
```

```
spec USER = USER_STATUS[user/user_status],
```

where *USER_STATUS* has been defined in Example 13.8; the latter is a static algebraic specification, and thus may be considered as a particular **RL** specification without proper transitions.

```
spec SYSTEM =
  enrich BUFFER + USER by
  sorts  system
  opns    $- \mid \_$ :  $buffer \times user \rightarrow system$ 
  rl     START:  $Empty \mid Initial \longrightarrow Empty \mid Putting_0$ 
  rl      $\tau_3$ :    $b \mid Putting_0 \longrightarrow Put(0, b) \mid Putting_1$ 
  rl      $\tau_4$ :    $b \mid Putting_1 \longrightarrow Put(1, b) \mid Reading$ 
  rl      $\tau_5$ :    $Not\_Empty(b) \Rightarrow$ 
                 $b \mid Reading \longrightarrow Remove(b) \mid Read(First(b))$ 
  rl     OK:     $b \mid Read(0) \longrightarrow b \mid Terminated$ 
  rl     ERROR:  $NotEq(i, 0) \Rightarrow b \mid Read(i) \longrightarrow b \mid Terminated$ 
```

Notice the major differences compared with the LTL specification of Example 13.8, even though the two seem very similar. In this case the specification is structured by giving first the specifications of the two components; but now the activity of the user is not given in *USER*, and only part of that of the buffer is given in *BUFFER*. Indeed, if, e.g., we have the rule $Putting_0 \rightarrow Putting_1$ in *USER*, then the user in any case may perform such a transition also without synchronizing with the buffer.

Furthermore there is no provision for knowing that *START*, *OK*, and *ERROR* are different interactions with the environment, while τ_1, \dots, τ_5 correspond to internal activities; see [AR97b] for a detailed comparison.

13.4 Dynamic data-types (A4)

In recent literature various approaches have been proposed to extend the classical algebraic framework for the specification of data types to handle processes; the first one was Goguen and Meseguer's reflexive semantics for object-orientation in [GM87]. All these approaches share some common features, which have been nicely summarized in [EO94] by Ehrig and Orejas, where they report informally a general schema for building an algebraic framework following the state-as-algebra style:

- states-as-algebras; thus, and implicitly, dynamics is modeled by a (labeled or not) transition system;
- all the statealgebras extend a fixed algebra of basic nondynamic values (static or value algebra);
- the elements of the carriers of the nonstatic sorts of a statealgebra are the components of the system at that moment, and the nonstatic operations represent how they are organized at that moment;
- state-transformations = transitions from a statealgebra to another statealgebra, correspond in most cases to operations of a special kind (dynamic operations) and in general are not homomorphisms (the organization and the number of components may change).

Notice that dynamic operation calls are the common mechanism to express the interactions with the environment; but in this way the reaction to an external stimulus (a dynamic operation call) must be deterministic (except if we leave the classical algebraic frameworks for some nondeterministic framework).

While the usual algebraic techniques may be used to define/specify the value and the statealgebras, there is no standard way to define/specify the dynamic operations (the transitions).

- there is no a general way to handle concurrency/distribution/cooperation among process components or to give in a structured way the specification of a concurrent system by composing the specifications of its components; usually each approach offers ad hoc techniques;

- most of these approaches have been developed with object-orientation in mind rather than concurrency generally.

Here, we briefly report on only some approaches; other can be found, for instance, in [Bau95,PP95]. Among them, evolving algebras are peculiar; since the emphasis is not on the data structure aspects, but more on the operational idea of state transitions; indeed, recently they have been called “Abstract State Machines” (ASM).

13.4.1 Evolving algebras (abstract state machines)

The basic idea of the “evolving algebras” (see, for instance, [Gur93,Gur95]) is perfectly summarized by their name. Essentially an evolving algebra (specification) consists of a description of a (nonlabeled) transition system, whose states are algebras on the same homogeneous signature built over the same universe (including Boolean values). Some of the operation symbols are qualified as “static” and their interpretation is the same in any (algebra which is a) state. The transitions are defined by rules of the following form:

$$econd \Rightarrow up_1, \dots, up_k$$

where, for each $j = 1, \dots, k$, the function update up_j has form

$$f_j(e_1^j, \dots, e_{n_j}^j) := e^j,$$

$econd, e_1^1, \dots, e_{n_1}^1, e^1, \dots, e_1^k, \dots, e_{n_k}^k, e^k$ are “descriptions” (any possible mathematically intelligible expressions) of elements of the universe, the first describing a Boolean value, and for $j = 1, \dots, k$, f_j is an operation of the signature of arity n_j .

The interpretation of one of such rules is that the system in a state A such that $econd$ holds on it, may pass to another algebra B where for each operation of the signature f , $f_B(x_1, \dots, x_n) =$

- the interpretation of e^j in A, if $f = f_j$ and x_1, \dots, x_n coincide with the interpretations of $e_1^j, \dots, e_{n_j}^j$ in A;
- $f_A(x_1, \dots, x_n)$, otherwise.

Obviously, the function updates up_1, \dots, up_k in a rule must be consistent in the sense that they do not simultaneously update the same function on the same arguments with different values.

Here, for simplicity, we consider only the basic evolving algebras, where there is no provision for concurrent structuring; concerning reactivity, some operations of the signature are classified “external” with the idea that they can change under the influence of (the nonfurther qualified) environment and cannot be modified by the rules.

Thus, essentially, to give an evolving algebra specification means to give a (nonlabeled) transition system where the states are finite tuples, which can

also be of functional type, describing the transitions by saying only which components of the state tuples change and how, and for the components of functional type, only for which arguments change and how. In this way state transformations are described in a very economic way. Consider, for example, the conditional rules describing the modification of tuple-like states, following, e.g., an LTL-style (see Section 13.3.1), as

$$econd \Rightarrow \langle c_1, \dots, c_i, \dots, c_n \rangle \rightarrow \langle c_1, \dots, c'_i, \dots, c_n \rangle$$

and

$$econd \Rightarrow \langle c_1, \dots, c_i, \dots, c_n \rangle \rightarrow \langle c_1, \dots, \lambda x. \text{ if } x = a \text{ then } b \text{ else } c_i(x), \dots, c_n \rangle$$

versus the corresponding simpler versions given in an evolving algebra style:

$$econd \Rightarrow C_i := c'_i \quad \text{and} \quad econd \Rightarrow C_i(a) := b.$$

After all, evolving algebras are nothing but lts where the states are algebras. As for other purely lts-based approaches, the treatment of liveness conditions and modular composition are less elegant.

Example 13.11. (Bit using evolving algebras)

```

** SIGNATURE
** Static operations
   Empty (0-ary)
   Remove, Dup, First, Not_Empty (1-ary)
   Put (2-ary)
   Initial, Terminated, Putting_0, Putting_1, Reading (0-ary)
   Read (1-ary)
   START, OK, ERROR (0-ary)
** Dynamic operations
   Buf_Cont, User_State, Output (0-ary)
** External operations
   Input (0-ary)
** definition of static operations
   ....
** RULES

Input = START  $\wedge$  Buf_Cont = Empty  $\Rightarrow$ 
   User_State := Putting_0

User_State = Putting_0  $\Rightarrow$ 
   User_State := Putting_1
   Buf_Cont := Put(0, Buf_Cont)

User_State = Putting_1  $\Rightarrow$ 
   User_State := Reading
   Buf_Cont := Put(1, Buf_Cont)

```

$$\begin{aligned} User_State = Reading \wedge Not_Empty(Buf_Cont) \Rightarrow \\ User_State := Read(First(Buf_Cont)) \\ Buf_Cont := Remove(Buf_Cont) \end{aligned}$$

$$true \Rightarrow Buf_Cont := Put(10^{10}, Empty)$$

$$true \Rightarrow Buf_Cont := Dup(Buf_Cont)$$

$$\begin{aligned} User_State = Read(0) \Rightarrow \\ User_State := Terminated \\ Output := OK \end{aligned}$$

$$\begin{aligned} User_State \neq Read(0) \Rightarrow \\ User_State := Terminated \\ Output := ERROR \end{aligned}$$

13.4.2 D-oids

D-oids [AZ95] are mathematical structures aimed at abstractly modeling concurrent systems by extending the algebraic approach for modeling data structures. In [AZ95] a very general approach is taken, since the definition of a d-oid is parameterized by the underlying static framework for (values and) statealgebras, but here for simplicity, we fix such framework to the usual algebras.

A d-oid has a signature, called a *dynamic signature*. In general a dynamic signature is a pair consisting of a signature Σ with a set of sorts S and a family of dynamic operation symbols DOP over S . A dynamic operation dop may have a functionality $dop: w \rightarrow [s]$, with $w \in S^*$ and $[s] \in S \cup \{A\}$. This corresponds to the idea that a dynamic operation may also return a null value. There are also constant dynamic operations dop having functionality $dop: \rightarrow [s]$.

A d-oid over $\langle \Sigma, DOP \rangle$ consists of a class \mathcal{A} of Σ -algebras and an interpretation of the dynamic operations. If $dop: w \rightarrow [s]$, then an interpretation of dop is a partial function mapping $\langle A, \bar{a} \rangle$, with $A \in \mathcal{A}$ and $\bar{a} \in |A|_w$, to a transformation of A into an algebra $B \in \mathcal{A}$ and a returned value $v \in |B|_{[s]}$, when $[s]$ is not null. A *transformation* of A in B is a triple $\langle A, f, B \rangle$, where f is a partial map from the carrier of A into the carrier of B , called a *tracking map*. The tracking map is essential for keeping track of the identity of the elements of the system: if e is an element in A , then we can recover it in the new state B by applying f to e ; tracking maps allow us to deal in a very abstract way with object creation (nonsurjective maps) and deletion (non total tracking maps). Tracking maps may be noninjective, to model the cases where some elements with different identities are glued together.

Static basic values may be provided by a value part, just an algebra, see [AZ95].

The interpretation of a constant dynamic operation $dop: \rightarrow [s]$ is an algebra $A \in \mathcal{A}$ and a returned value $v \in |A|_{[s]}$, whenever $[s]$ is not null.

Finally, it is possible to extend in a natural way the notion of term to the dynamic case. Terms define a syntactic structure, the *term d-oid*, which is, under some assumptions, a free structure for the appropriate categorical setting [AZ96].

Concerning specifications based on d-oids, [Zuc96] presents a general way to build institutions for dynamic data type specifications and shows an application to the d-oid case. The formulas of the proposed logic allow us to express static properties, precisely on the value part and the statealgebras (a kind of system invariants). Concerning the properties on dynamics, it allows formulas for expressing pre- and post-conditions on sequential compositions of dynamic operations, represented by the elements of the term d-oid, and for requiring that two compositions are the same, i.e., they represent the same transformation.

Example 13.12. (Bit using d-oids) The value part is given by the initial model of following algebraic specification, where *USER_STATUS* and *MESSAGE* have been given respectively in Examples 13.8 and 13.3, and *INT_QUEUE* is in Appendix A.

```
spec VALUE = INT_QUEUE + USER_STATUS + MESSAGE

spec SYSTEM =
  enrich VALUE by
    ** value part
    ** statealgebra part
    Buf_Cont: → queue
    User_State: → user_status
    Output: → message
    ** dynamic operations
    START, INT1, INT2, INT3: →
    OUTPUT: → message
    ** dynamic properties
    {Buf_Cont = Empty ∧ User_State = Initial} START
    {User_State = Putting_0}
    {User_State = Putting_0} INT1
    {User_State = Putting_1 ∧ Buf_Cont = Put(0, Buf_Cont)}
    {User_State = Putting_1} INT1
    {User_State = Reading ∧ Buf_Cont = Put(1, Buf_Cont)}
    {User_State = Reading ∧ Not_Empty(Buf_Cont)} INT1
    {User_State = Read(First(Buf_Cont)) ∧
    Buf_Cont = Remove(Buf_Cont)}
    INT2 {Buf_Cont = Put(1010, Empty)}
    INT3 {Buf_Cont = Dup(Buf_Cont)}
    {User_State = Read(0)} INT1
    {User_State = Terminated ∧ Output = OK}
    {User_State = Read(i) ∧ ¬(i = 0)} INT1
    {User_State = Terminated ∧ Output = ERROR}
    {User_State = Terminated} m ← OUTPUT
```


13.4.3 Algebraic specifications with implicit state

The specification formalism of Dauchy and Gaudel [DG93,GDK96] is based on the notion of *elementary access function* and *elementary modifier*. Elementary access functions characterize the structure of the states of the system, as kinds of observation functions, while elementary modifiers allow us to perform updates of the elementary accesses without returning any value. Elementary modifiers are built-in features of the specification language, associated with the elementary access functions.

A specification in this formalism is a 4-tuple

$$\langle \langle \Sigma, AX \rangle, \langle \Sigma_{ac}, AX_{ac} \rangle, \langle \Sigma_{mod}, Def_{mod} \rangle, AX_{init} \rangle,$$

where:

- $\langle \Sigma, AX \rangle$ is the specification of the static values used.
- $\langle \Sigma_{ac}, AX_{ac} \rangle$ is the specification of the access functions and is a conservative extension (see, Chapter 6) of $\langle \Sigma, AX \rangle$ with no new sort (thus $\Sigma \subseteq \Sigma_{ac}$). Some of the access functions are elementary, while the others are defined in terms of the elementary ones by the access axioms of AX_{ac} .
- The admissible initial states are characterized by the set of axioms AX_{init} .
- The definition of the elementary access functions makes implicitly available the corresponding elementary modifiers in the following way: given an elementary access function f with functionality $s_1 \times \dots \times s_n \rightarrow s$, the corresponding elementary modifier is μ - f with domain $s_1 \times \dots \times s_n \times s$. Elementary modifiers are the tools for describing the statealgebras transformation.

Given some terms with variables t_1, \dots, t_n of sorts s_1, \dots, s_n and a term t of sort s , the meaning of the statement μ - $f(t_1, \dots, t_n, t)$ is a modification of f . More precisely, it transforms a statealgebra A into a statealgebra B such that:

- * $f_B(v_1, \dots, v_n) = (\sigma(t))^B$ if there exists a ground substitution σ such that for $i = 1, \dots, n$ $v_i = (\sigma(t_i))^A$;
- * $f_B(v_1, \dots, v_n) = f_A(v_1, \dots, v_n)$ otherwise;
- * derived access functions which depend on f are changed accordingly;
- * any other operation and all carriers are unchanged.

$\langle \Sigma_{mod}, Def_{mod} \rangle$ define some composite modifiers, the functionalities of which have no range. The axioms in Def_{mod} are positive conditional and their premises are built on Σ_{ac} . They define the modifiers using statements built from the elementary modifiers and the following constructs: **nil** identity; **;** sequential composition; **and** composition in any order (it is responsibility of the specifier to check that the result of the composition does not depend on the order); **•** denotes modifications made on the same state, i.e., all preconditions and arguments of the involved modifiers must be evaluated in the initial state prior to doing all corresponding modifications.

Example 13.13. (Bit using access functions and modifiers) Let *VALUE* be the algebraic specification defined in Example 13.12.

```

spec SYSTEM =
  enrich VALUE by
    ** basic static values
    ** elementary accesses
    Buf_Cont: → queue
    User_State: → user_status
    Output: → message
    ** modifiers
    START: →
    τ: →
    ** modifier definitions
    User_State = Initial ∧ Buf_Cont = Empty ⇒
      START = μ-User_State(Putting_0)

  τ =
  cases
  User_State = Putting_0 ⇒
    (μ-User_State(Putting_1) ∧ μ-Buf_Cont(Put(0, Buf_Cont)))
    ∨ (μ-Buf_Cont(Put(1010, Empty)))
    ∨ (μ-Buf_Cont(Dup(Buf_Cont)))
  User_State = Putting_1 ⇒
    (μ-User_State(Reading) ∧ μ-Buf_Cont(Put(1, Buf_Cont)))
    ∨ (μ-Buf_Cont(Put(1010, Empty)))
    ∨ (μ-Buf_Cont(Dup(Buf_Cont)))
  User_State = Reading ∧ Not_Empty(Buf_Cont) ⇒
    (μ-User_State(Read(First(Buf_Cont))) ∧
    μ-Buf_Cont(Remove(Buf_Cont)))
    ∨ (μ-Buf_Cont(Put(1010, Empty)))
    ∨ (μ-Buf_Cont(Dup(Buf_Cont)))
  User_State = Read(0) ⇒
    (μ-User_State(Terminated) ∧ μ-Output(OK))
    ∨ (μ-Buf_Cont(Put(1010, Empty)))
    ∨ (μ-Buf_Cont(Dup(Buf_Cont)))
  User_State = Read(i) ∧ NotEq(i, 0) = True ⇒
    (μ-User_State(Terminated) ∧ μ-Output(ERROR))
    ∨ (μ-Buf_Cont(Put(1010, Empty)))
    ∨ (μ-Buf_Cont(Dup(Buf_Cont)))
  end cases

```

13.4.4 Statealgebras with references

In [GR95,GR97] Große-Rhode presents a state-as-algebra approach based on a general idea of “reference”.

A statealgebra is a partial algebra which is an extension of a given algebra *A*, called the *base model* (static value algebra). More precisely, a state is considered as a static data type where references are added. For some

sorts, say s , a special reference sort $ref(s)$ and a contents operation symbol $!_s : ref(s) \rightarrow s$ are introduced in the signature. The base model is a model of a normal partial equational specification, called the *base specification*, with existence equations and some minor restrictions concerning equations between references.

For a list of (pairwise different) references $d_\omega = d_1 \dots d_n$ with $d_i \in |A|_{ref(s_i)}$ and a corresponding list of values $a_\omega = a_1 \dots a_n$ with $a_i \in |A|_{s_i}$, the *state* $A^{[d_\omega := a_\omega]}$ on the base model A , where the references d_i have contents a_i , is formally defined as a free extension of A by an existence equation of the form $!_{s_1}(d_1) = a_1 \wedge \dots \wedge !_{s_n}(d_n) = a_n$.

The definition of states as free extensions of the base model allows one to formalize the notion of *persistent state*: a persistent state on A is a model of the base specification whose restriction to the signature of the value datatypes (disregarding the reference sorts) is isomorphic to A . Intuitively a persistent state can be regarded as an extension with the content functions of A , i.e., as a pair $\langle A, env \rangle$, where env is an environment which is a family of partial functions mapping references to values. It is proved that persistent states are in a one-to-one correspondence with the pairs $\langle A, env \rangle$.

On top of a base specification a *transition specification* can be defined. Dynamic operations are specified by a set of method definitions, which are conditional parallel assignments. It is possible to have several assignments for the same method with overlapping conditions, hence dynamic operations are nondeterministic. Finally, state transitions are specified by method expressions built by the application of methods to arguments and by sequential composition of them.

Some interesting results concern structured specifications; following the well-established theory of composition of specifications and of parametric specifications in an arbitrary institution, Große-Rhode proves that his specifications enjoy the properties needed for defining the usual structuring mechanisms for composing specifications.

Example 13.14. (Bit using statealgebras with references) Let *VALUE* be the specification defined in Example 13.12.

```

spec VALUE-REF =
  enrich VALUE[buffer/queue] by
  sorts   ref(user_status), ref(buffer), ref(message)
  refs    U : → ref(user_status)
          B : → ref(buffer)
          Output : → ref(message)

spec SYSTEM =
  enrich VALUE-REF by
  ** methods
  START : → ref(user_status)
  !U = Initial ∧ !B = Empty ⇒ START(U) := Putting_0.O

```

$$\begin{aligned}
&INT1: \rightarrow \text{ref}(\text{buffer}) \times \text{ref}(\text{user_status}) \\
&!U = \text{Putting_0} \Rightarrow INT1(B, U) := \langle \text{Put}(0, !B), \text{Putting_1} \rangle . O \\
&!U = \text{Putting_1} \Rightarrow INT1(B, U) := \langle \text{Put}(1, !B), \text{Reading} \rangle . O \\
&!U = \text{Reading} \wedge \text{Not_Empty}(!B) \Rightarrow \\
&\quad INT1(B, U) := \langle \text{Remove}(!B), \text{Read}(\text{First}(!B)) \rangle . O
\end{aligned}$$

$$\begin{aligned}
&INT2: \rightarrow \text{ref}(\text{buffer}) \\
&INT2(B) := \text{Put}(10^{10}, \text{Empty}) \\
&INT2(B) := \text{Dup}(!B)
\end{aligned}$$

$$\begin{aligned}
&INT3: \rightarrow \text{ref}(\text{user_status}) \times \text{ref}(\text{message}) \\
&!U = \text{Read}(0) \Rightarrow INT3(U, \text{Output}) := \langle \text{Terminated}, \text{OK} \rangle . O \\
&!U = \text{Read}(i) \wedge \text{NotEq}(i, 0) \Rightarrow \\
&\quad INT3(U, \text{Output}) := \langle \text{Terminated}, \text{ERROR} \rangle . O
\end{aligned}$$

13.5 Conclusion

We have distinguished four main approaches in the use of algebraic techniques for the specification of reactive, and concurrent systems and presented some methods illustrative of the different viewpoints. There has not been room to include all methods, in particular those more recent (like the coalgebraic) or requiring a deeper treatment (like hidden specifications), but we have provided pointers to the relevant literature. Nor we have made any attempt at comparing the different methods, since a thorough comparison should follow some rigorous criteria, which are still under discussion (see, e.g., some hints in [AR97a]). Instead we have provided a guided tour, hopefully stimulating further reading and research.

Looking back, we can now observe that, in order to handle properly the features typical of concurrent and reactive systems, the algebraic techniques need some kind of extension of a very different nature. First they all need an underlying model able to deal with the concurrency issues (like Petri nets or Labeled Transition Systems). Then there are specific adjustments either at the level of the specification language (**A2**), or of some basic technical point (generalized bisimulations, coalgebras instead of algebras, hidden specifications), or at the method level [BCPR96].

In general any really usable formalism for the specification of systems must be complemented by a specification formalism for data and in this respect algebraic techniques have the advantage of being very abstract and linked to languages supporting modularity. This is the rationale behind the success of methods following viewpoint **A2**, like LOTOS.

We can also observe that only the algebraic methods following approach **A3** keep the fully axiomatic flavor of the original algebraic specifications; this would apply to the hidden specification and coalgebraic methods too.

An issue which has only been mentioned as an aside, but of paramount importance, is the support of automatic tools both for development and verification. This is a fast developing field, which could provide one basic key

to the successful use of algebraic techniques in the future. Another key could come from a standardization of the algebraic notation and of the associated methods, contrary to the direction of the current proliferation of notations. In this respect we are greatly looking forward to the outcome of CoFI, the ongoing Common Framework Initiative, sponsored by the IFIP WG 2.2 [Mos97].

Acknowledgement. We thank Davide Ancona and Elena Zucca for Section 13.4.

Work partially funded by HCM project Express and MURST 40% Modelli della computazione e dei linguaggi di programmazione.

A Specifications of data types used by Bit

Below we report the algebraic specifications of the data types, integers and finite queues of integers, which are used in various specifications of **Bit**.

```

spec INT =
  sorts   int
  opns    0: → int
          S, P: int → int
          _ + _: int × int → int
  preds   _ < _: NotEq: int × int
  axioms  0 + i = i
          S(i) + i' = S(i + i')
          P(i) + i' = P(i + i')
          0 < S(0)
          P(0) < 0
          i < i' ⇒ (S(i) < S(i') ∧ P(i) < P(i') ∧ i < S(i') ∧ P(i) < i')
          (i < i' ∨ i' < i) ⇒ NotEq(i, i')

spec INT_QUEUE =
  enrich INT by
  sorts   queue
  opns    Empty: → queue
          Put: int × queue → queue
          First: queue → int
          Remove, Dup: queue → queue
          ** Dup duplicates the first element of a queue
  preds   Not_Empty: queue
  axioms  Not_Empty(Put(i, q))
          First(Empty) = 0
          First(Put(i, Empty)) = i
          First(Put(i, Put(i', q))) = First(Put(i', q))
          Remove(Empty) = Empty
          Remove(Put(i, Empty)) = Empty
          Remove(Put(i, Put(i', q))) = Put(i, Remove(Put(i', q)))
          Dup(Empty) = Empty
          Dup(Put(i, Empty)) = Put(i, Put(i, Empty))
          Dup(Put(i, Put(i', q))) = Put(i, Dup(Put(i', q)))

```

References

- [AGR92] E. Astesiano, A. Giovini, and G. Reggio. Observational structures and their logic. *Theoretical Computer Science*, 96(1):249–283, 1992.
- [AMRW85] 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*, volume 185 of *Lecture Notes in Computer Science*. Springer, 1985.
- [AR87] 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*, volume 249 of *Lecture Notes in Computer Science*. Springer, 1987.
- [AR93] E. Astesiano and G. Reggio. Specifying reactive systems by abstract events. In *Proc. 7th Intl. Workshop on Software Specification and Design (IWSSD-7)*, Los Alamitos, CA, 1993. IEEE Computer Society.
- [AR96a] 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*, volume 1169 of *Lecture Notes in Computer Science*. Springer, 1996.
- [AR96b] E. Astesiano and G. Reggio. Labelled transition logic: An outline. Technical Report DISI-TR-96-20, DISI – Università di Genova, Italy, 1996.
- [AR97a] E. Astesiano and G. Reggio. Formalism and method. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*. Springer, 1997.
- [AR97b] 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.
- [AZ95] E. Astesiano and E. Zucca. D-oids: a model for dynamic data types. *Mathematical Structures in Computer Science*, 5(2):257–282, 1995.
- [AZ96] E. Astesiano and E. Zucca. A free construction of dynamic terms. *Journal of Computer and System Sciences*, 52(1), 1996.
- [Bau95] H. Baumeister. Relations as abstract datatypes: An institution to specify relations between algebras. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Proc. TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science*. Springer, 1995.
- [BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14, 1987.
- [BCM88] E. Battiston, F. De Cindio, and G. Mauri. OBJSA nets: a class of high-level nets having objects as domains. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 340 of *Lecture Notes in Computer Science*. Springer, 1988.
- [BCPR96] M. Bidoit, C. Chevenier, C. Pellen, and J. Ryckbosh. An algebraic specification of the steam-boiler control system. In J.-R. Abrial, E. Borger, and H. Langmaack, editors, *Formal Methods for Industrial Applications*, volume 1165 of *Lecture Notes in Computer Science*. Springer, 1996.
- [BHK89] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series. Addison-Wesley, 1989.

- [BK86] J.A. Bergstra and J.W. Klop. Process algebra: Specification and verification in bisimulation semantics. In M. Hazewinkel, J.K. Lenstra, and L.G.L.T. Meertesen, editors, *Math. & Comp. Sci. II*, volume 4 of *CWI Monograph*. North Holland, 1986.
- [Bro87] M. Broy. Semantics of finite or infinite networks of communicating agents. *Distributed Computing*, 2, 1987.
- [Bro93] M. Broy. Interaction refinement: The easy way. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI Series, Series F: Computer and System Sciences*. Springer, 1993.
- [Bro96] M. Broy. Specification and refinement of a buffer of length one. In M. Broy, editor, *Deductive Program Design*, volume 152 of *NATO ASI Series, Series F: Computer and System Sciences*. Springer, 1996.
- [Bro97] M. Broy. Compositional refinement of interactive system. *Journal of the ACM*, 44(6), 1997.
- [BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.
- [CR96] E. Coscia and G. Reggio. Deontic concepts in the algebraic specification of dynamic systems: The permission case. In O.-J. Dahl, M. Haveran, and O. Owe, editors, *Recent Trends in Data Types Specification, Proc. 11th Workshop on Specification of Abstract Data Types joint with the 8th General COMPASS Meeting*, volume 1130 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 1996.
- [CR97] G. Costa and G. Reggio. Specification of abstract dynamic data types: A temporal logic approach. *Theoretical Computer Science*, 173(2), 1997.
- [Dav65] M. Davis, editor. *The Undecidable*. Raven Press, New York, 1965.
- [Den80] J.B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11), 1980.
- [DG93] P. Dauchy and M.-C. Gaudel. Implicit state in algebraic specifications. In U.W. Lipeck and G. Koschorreck, editors, *Proc. Intl. Workshop Is-Core'93, Hannover (Germany)*. Universität Hannover, 1993.
- [DH91] C. Dimitrovici and U. Hummert. Composition of algebraic high-level nets. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification, Proc. 7th Workshop on Specification of Abstract Data Types*, volume 534 of *Lecture Notes in Computer Science*. Springer, 1991.
- [EFH83] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An algebraic specification language with two levels of semantics. Technical Report 83-01, Technische Universität Berlin, 1983.
- [Eme90] A.E. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*. Elsevier Science Publishers B.V. (North Holland), 1990.
- [EO94] H. Ehrig and F. Orejas. Dynamic abstract data types: An informal proposal. *EATCS Bulletin*, 53, 1994.
- [ES95] H.-D. Ehrich and A. Sernadas. Local specification of distributed families of sequential objects. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification: 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop*, volume 906 of *Lecture Notes in Computer Science*, pages 218–235. Springer, 1995.

- [GD94] J.A. Goguen and R. Diaconescu. Towards an algebraic semantics for the object paradigm. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification*, volume 785 of *Lecture Notes in Computer Science*, pages 1–29. Springer, 1994.
- [GDK96] M.-C. Gaudel, P. Dauchy, and C. Khoury. A formal specification of the steam-boiler control problem by algebraic specification with implicit state. In J.-R. Abrial, E. Borger, and H. Langmaack, editors, *Formal Methods for Industrial Applications*, volume 1165 of *Lecture Notes in Computer Science*. Springer, 1996.
- [GG94] Dov Gabbay and Franz Guenther, editors. *What is a Logical System?* Oxford University Press, 1994.
- [GM87] J.A. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editor, *Research Directions in Object-Oriented Programming*, Computer System Series, pages 417–477. MIT Press, 1987.
- [GR95] M. Große-Rhode. *Specification of Transition Categories – An Approach to Dynamic Abstract Data Types*. PhD thesis, Technische Universität Berlin, 1995.
- [GR97] M. Große-Rhode. Transition specifications for dynamic abstract data types. *Applied Categorical Structures*, 5, 1997.
- [Gur93] Y. Gurevich. Evolving algebras, an attempt to discovery semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*. World Scientific, 1993.
- [Gur95] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
- [Hoa85] C.A.R. Hoare, editor. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [I.S89] 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.
- [JR91] K. Jensen and G. Rozenberg, editors. *High-Level Petri Nets – Theory and Application*. Springer, 1991.
- [JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
- [Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Information Processing 77*. North-Holland, 1974.
- [Lam83] L. Lamport. Specifying concurrent program modules. *ACM TOPLAS*, 3, 1983.
- [LOT97] *Enhanced LOTOS Documentation*, 1997. Available at <ftp://ftp.dit.upm.es/pub/lotos/elotos/>.
- [Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
- [Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
- [Mil89] R. Milner. *Communication and concurrency*. Prentice Hall, London, 1989.

- [MM93] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, Computer Science Laboratory, SRI International, 1993.
- [Mos97] Peter D. Mosses. CoFl: The common framework initiative for algebraic specification and development. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 1997.
- [MP89] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*. Springer, 1989.
- [MV89] S. Mauw and G.J. Veltink. An introduction to psf_a . In J. Diaz and F. Orejas, editors, *Proc. TAPSOFT'89, Vol. 2*, volume 352 of *Lecture Notes in Computer Science*, pages 375–389. Springer, 1989.
- [MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII, 1990.
- [MV91] S. Mauw and G.J. Veltink. A proof assistant for PSF. In K. Larsen and A. Skou, editors, *Proc. 3rd Workshop on Computer Aided Verification, Vol. 1*, Aalborg, Denmark, 1991. The University of Aalborg.
- [MV93] S. Mauw and G.J. Veltink. *Algebraic Specification of Communication Protocols*, volume 36 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Aalborg, Denmark, 1993.
- [NH84] R. De Nicola and M. Hennessy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.
- [Par81] D. Park. Concurrency and automata on infinite sequences. In *Proc. 5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*. Springer, 1981.
- [Plo83] Gordon D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Formal Description of Programming Concepts – II, Proc. IFIP TC-2 Working Conference*, pages 199–225. North-Holland, 1983.
- [Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, New York, 1977. IEEE Computer Society Press.
- [Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*. Springer, 1986.
- [PP95] F. Parisi-Presicce and A. Pierantonio. Dynamical behaviour of object systems. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification: 10th Workshop on Specification of Abstract Data Types – Selected Papers*, volume 906 of *Lecture Notes in Computer Science*. Springer, 1995.
- [PSF97] PSF toolkit manual pages. Technical report, WINS, University of Amsterdam, 1997. Available at <http://www.wins.uva.nl/~bobd/work/>.
- [Reg91] 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, Proc. 7th Workshop on Specification of Abstract Data Types*, volume 534 of *Lecture Notes in Computer Science*. Springer, 1991.

- [Reg93] G. Reggio. Event logic for specifying abstract dynamic data types. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification, Proc. Workshop on Specification of Abstract Data Types ADT'91*, volume 655 of *Lecture Notes in Computer Science*. Springer, 1993.
- [Rei85] W. Reisig. *Petri Nets: an Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.
- [Rei91] W. Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80(1):1–34, 1991.
- [Rei98] W. Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer, 1998.
- [RT86] G. Rozenberg and P.S. Thiagarajan. Petri nets: Basic notions, structure, behaviour. In *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*. Springer, 1986.
- [Sti92] C. Stirling. Modal and temporal logics. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 2*, pages 477–563. Clarendon Press, Oxford, 1992.
- [Vau87] J. Vautherin. Parallel system specifications with coloured petri nets and algebraic data types. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 266 of *Lecture Notes in Computer Science*. Springer, 1987.
- [vE91] P. van Eijk. Tools for LOTOS, a lotosfere overview. Memoranda Informatica 91-25, Universiteit Twente - Faculteit der Informatica, Enschede, 1991.
- [vG90] R.J. van Glabbeek. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proc. CONCUR'90, Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*. Springer, 1990. Extended abstract.
- [Zuc96] E. Zucca. From static to dynamic abstract data types. In W. Penczek and A. Szlas, editors, *Proc. MFCS'96*, volume 1113 of *Lecture Notes in Computer Science*. Springer, 1996.