Algebraic Specification of Concurrency*

Egidio Astesiano and Gianna Reggio

DISI
Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova – Italy
{ astes , reggio } @ cisi.unige.it

Introduction

Let us first summarize what algebraic specification is about, following [66]. Algebraic specification methods provide techniques for data abstraction and the structured specification, validation and analysis of data structures.

Classically, a (concrete) data structure is modelled by a many-sorted algebra (possibly term-generated); various categories of many-sorted algebras can be considered, like total, partial, order-sorted, with predicates and so on. An isomorphism class of data structures is called an *abstract data type* (shortly *adt*) and an *algebraic specification* is a description of one or more abstract data types. There are various approaches for identifying classes of abstract data types associated with an algebraic specification, which constitute its semantics: initial, terminal, observational; a semantics is *loose* when it identifies a class (usually infinite) of adt's.

Since data structures can be very complex (a flight reservation system, e.g.), *structuring* and *parameterization* mechanisms are fundamental for building large specifications.

Together with a rigorous description of data structures, algebraic specifications support stepwise refinement from abstract specifications to more concrete descriptions (in the end, programs) of systems by means of the notion of *implementation* and techniques for proving *correctness* of implementations. In this respect formal *proof systems* associated with algebraic specifications play a fundamental role. Finally *specification languages* provide a linguistic support to algebraic specifications.

The purpose of the algebraic specification of concurrent systems is to specify structures where some data represent processes or states of processes, i.e. objects about which it is possible to speak of dynamic evolution and interaction with other processes; more generally we can consider as the subject of algebraic specification of concurrency those structures able to describe entities which may be active participants of events. Such data structures will be called simply "concurrent systems", where "concurrent" conveys different meanings, from "occurring together" to "compete for the same resources" and to "cooperate for achieving the same aim".

The aim of this paper is twofold: to analyse the aims and the nature of the algebraic specifications of concurrency and to give, as examples, a short overview of some (not all) relevant work.

* This work has been supported by COMPASS-Esprit-BRA-W.G. the project MURST 40% "Metodi e specifiche per la concorrenza" and by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of C.N.R. (Italy).

In Sect. 1 we introduce some basic concepts and terminology about processes: the various models around which the specification models are built and the fundamental issues of (observational) semantics, formal description and specification; moreover we give some illustrative examples of specification problems to be used later for making more concrete some general considerations and for assessing different methods.

In Sect. 2 we try to qualify the field: indicating three different fundamental motivations/viewpoints (and distinguishing between methods and instances); then outlining the issues to deal with; finally illustrating by two significant examples/approaches how this field stimulates innovations and improvements beyond the classical theory of adt's.

In Sect. 3 we outline some relevant approaches; the presentation is related to the issues discussed in Sect. 2 and the specification examples presented in Sect. 1. However, being impossible to report on all methods, we have mainly used some approaches to illustrate, as examples, concrete ways of tackling the issues of the field..

1 Processes and Concurrent Systems

Informally a process is an entity with the capability of performing an activity within which it may interact with other entities and/or with the environment. The interaction may consist in communicating, synchronizing, cooperating, acting in parallel, competing for resources with other processes and/or with the environment.

By a concurrent system we informally mean a process consisting of component processes that are operating concurrently.

We are of course interested in those aspects of processes that support the design and implementation of software systems. Thus we are looking for a formal support to the specification, programming, implementation and verification phases. For this it is crucial to have good models for processes.

Now, the usual formal model of a sequential software system (program) as input/output, or state to state, function is no longer adequate for processes. Moreover, to date no single model seems to capture all the relevant formal aspects of a process. Hence in the following we will briefly introduce the most significant formal models which have been used in connection with the algebraic specification of processes. In the meantime we introduce some terminology typical of concurrency, which will be useful in the sequel.

Warning. We are presenting basic models and not formalisms; for example, labelled transition systems (and variations) are the common basic model for different formalisms like CCS, CSP, MEIJE,Π-calculus, etc. Moreover some formalisms, notably the many variations of Petri nets, allow to represent different aspects of systems and provide a variety of basic models. However our aim here is only to give the non-expert in concurrency some very basic information, in order to understand the following presentation of specification formalisms. It is not at all an overview of existing formalisms in concurrency.

1.1 Basic Models

Labelled Transition Systems. The use of labelled transition systems for modelling processes has been advocated mainly by Milner and Plotkin (see [50, 55]).

A *labelled transition system* (shortly *lts*) is a triple $(STATE, LABEL, \rightarrow)$, where $STATE$ and $LABEL$ are two sets and $\rightarrow \subseteq STATE \times LABEL \times STATE$. A triple $(s, l, s') \in \rightarrow$, also written $s \xrightarrow{l} s'$, means that the process modelled by the lts has the capability of passing from the state s into the state s' under an interaction with the external environment represented by the label l. In the simplest case, when the transition is purely internal to the system and there is no relationship with the environment, the label can be dropped or better represented by a special element, which is usually written τ (as in CCS) and called "silent move" or "internal action".

For example, the capabilities associated with a process executing an action of receiving a message, briefly denoted by $Rec(x, ch)$, assigned as value to a variable x on a channel ch could be represented by a set of labelled transitions of the form $s \xrightarrow{Rec(v,ch)} s'(v)$ (one for each v in the set of values that can be received). That means that the process can pass from the state s (corresponding to the situation immediately before the execution of $Rec(x, ch)$) into a state $s'(v)$ (which records that the value v has been received and assigned to the variable x) performing an action of receiving v from the outside along the channel ch.

The capabilities associated with a process executing the action of sending the message e on the channel ch, briefly denoted by $Send(e, ch)$, could be represented by the labelled transition $s \xrightarrow{Send(v_0,ch)} s'$. That means that the process can pass from the state s (corresponding to the situation immediately before the execution of $Send(e, ch)$) to the state s' performing the action of sending the value of the expression e in s (v_0) along the channel ch. Notice that an lts may have several different transitions starting from the same state and that allows us to represent the nondeterministic behaviour of processes.

We need also to model groups of interacting processes; in these cases we can use particular classes of lts's built from some component lts's. The states of the overall system have as subparts states of the component systems and its transitions are determined by the transitions of the component systems. For example the parallel composition p of two processes p_1 and p_2 (which interact by exchanging messages through channels) could be represented by a state $cs = \{s_1, s_2\}$, where s_1 and s_2 represent the initial states of the two processes p_1 and p_2. Assuming the transitions of the components $s_1 \xrightarrow{Rec(v_0,ch)} s'_1$ and $s_2 \xrightarrow{Send(v_0,ch)} s'_2$, the following transition of cs corresponds to the synchronized exchange of the value v_0 between p_1 and p_2: $cs \xrightarrow{\tau} \{s'_1, s'_2\}$ (the transition is labelled by τ since there is no need of further interaction with the world outside p).

Nondeterminism can also come from parallelism, for example when a process can perform some action with at least two other processes. Consider the process p_1 above in parallel with p_2 and with a process p_3 in a state s_3 having the capability $s_3 \xrightarrow{Send(v_0,ch)} s'_3$; clearly there are at least two exclusive possible transitions of the system consisting of the three processes in parallel. Specifically:

$\{s_1, s_2, s_3\} \xrightarrow{\tau} \{s'_1, s'_2, s_3\}$ and $\{s_1, s_2, s_3\} \xrightarrow{\tau} \{s'_1, s_2, s'_3\}$.

Lts's are very suitable for composing processes in a modular way; however they require a further procedure of abstraction for eliminating details and representing causality relationships between events. Indeed lts's are usually a too detailed description of processes, so that equivalent processes may be described by two different values (labelled trees with states) in an lts. Thus it is necessary to define semantics via equivalence classes and there is an extensive literature on the subject (see e.g. [37]). This issue is discussed in some more detail in Sect. 1.2 and, from a very general algebraic viewpoint, encompassing lts's in Sect. 2.3.

Event Structures. We may model processes by considering a set of notable facts which can happen during their activity called *events* (e.g., sending/receiving a value, changing/testing the content of a local storage) and then by describing the relevant relationships among them, as *causality* and *mutual exclusion*. In this way we can give a view of the processes more abstract than using lts's.

Formally a process is modelled by an *event structure* (see [65]), i.e. a triple $(E, \geq, \#)$, where E is a set (the events), $\geq$ is a partial order on E (causality relation) and $\#$ is binary relation on E (mutual exclusion relation).

It is important to note that using event structures we can describe in a simple way processes where two events are truly concurrent (i.e., where there is no causal/temporal relationship between them).

Consider, for example, the event structure ES graphically represented in Fig. 1; there the events e_1 and e_2 are truly concurrent: i.e. there is no relationship between the happening of e_1 with the happening of e_2 and vice versa; while e_3 may happen only after that e_1 and e_2 have happened; and e_2 and e_4 cannot happen simultaneously.

Fig. 1. An event structure ES

Unfortunately it is not easy to give a simple way for composing the event structures modelling the component processes of a complex concurrent system to get the structure modelling the whole system (see e.g. [64] which requires non-trivial categorical techniques).

Petri Nets. Petri nets are probably the oldest and best-known model of processes (see e.g. [59]). Starting from the original definition, nowadays many variations of Petri nets have been developed with different basic models. Here we briefly outline some typical features of this kind of models by considering a very basic one, which is now called Petri net (i.e. directed bipartite graph).

A *Petri net* is a directed graph $PN = (N, A)$, where the set of the nodes N is split into *places* and *transitions*, $N = P \cup T$, and the arcs connect only places to transitions and transitions to places. Given a transition t, the *premise* and the *consequence* of t are the following sets of places: $\{p \mid \text{there exists an arc from } p \text{ to } t\}$ and $\{p \mid \text{there exists an arc from } t \text{ to } p\}$ (notice that in general the two sets are not disjoint).

To describe the dynamic behaviour of Petri nets we consider marked nets, i.e. nets where each place is marked by a number of tokens. In a marked net a transition is enabled when all places in its premise have at least a token. An enabled transition may *fire* changing the marking of the net as follows: a token is eliminated by all the places in its premise and a token is added to all places in its consequence. Then the dynamic activity of a marked net is given by the firing of its enabled transitions; notice that in general more than one transition is enabled, so that a net can be used to model also nondeterministic processes. A marked net models a process in a particular situation, while the firing of the transitions describes its possible activities.

Petri nets are very popular since they allow to give nice graphical representations of the activity of the processes; however also in this case it is not very simple and natural to compose the nets describing the component processes of a complex system to get the net describing the whole system; to this end particular kinds of nets have been developed (see e.g. the superposed automata of [15]). For a survey on modular approaches to Petri nets see [16].

In Fig. 2 we report a simple example of a marked Petri net with three transitions and five places which describes a system transferring the tokens from the places IN1 and IN2 into the place OUT.

Fig. 2. A Petri net

Dataflows. The dataflow approach (see e.g. [41]) provides a completely different, but more specialized, model. The basic idea is to see a process as a box able to receive in an asynchronous way values along some input lines (or channels) and then to return other values along some output lines always in an asynchronous way. Here asynchronous means that values are received also if the process is performing some calculation on other values received previously and the output values are returned also if the receiver is not immediately ready to get them.

Thus a dataflow may be modelled by a function from tuples of (also infinite) sequences of values (those received on the input channels) into tuples of (also infinite) sequences of values (those returned on the output channels).

In this framework it is easy to compose several dataflows together: it is just as to compose functions.

This model is particularly apt to describe processes which interact in the above asynchronous way, while it is not very convenient for describing processes interacting synchronously.

In Fig. 3 we report a simple example of a dataflow network which describes a system receiving in input integer numbers on channel i and returning those which are positive, zero and negative respectively on channels p, z and n; the two component dataflows SEL_POS and SEL_NEG select respectively the positive and negative numbers.

Fig. 3. A dataflow network

1.2 Semantics

Whatever kind of model we choose, when describing processes we have still the problem of defining the right semantic equivalence, i.e. in general it is not true that two processes are semantically equivalent iff they have associated the same model. In this section we consider this problem only for the lts's, but the situation is analogous for the other models, for which other notions of semantic equivalence have been developed.

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

By associating with a process p the transition tree having as root the initial state of p we give an operational semantics: two processes are operationally equivalent whenever the associated transition trees are the same, see [51]. But usually such semantics is too fine, since it takes into account all details of the process activity. It may happen that two processes which we consider semantically equivalent have associated different trees. A simple case is when we consider the trees associated with two sequential processes (i.e., performing only sequential commands), represented by two states p and p', thus they perform only internal activities (i.e., no interactions with the external environment); the associated transition trees (reported in Fig. 4) are unary trees, with all the arcs labelled by the symbol of internal action τ. If we

Fig. 4. Transition trees associated with two sequential processes

consider an input-output semantics, then they are equivalent iff p, p' are equivalent w.r.t. the input and p_{F}, p'_{F} are equivalent w.r.t. the output; the differences about other aspects (intermediate states, number of the intermediate transitions, etc.) are not considered.

From this simple example, we understand also that we can get various interesting semantics on processes modelled by lts's depending on what we observe of them. For instance, consider the well-known strong bisimulation of Park ([53]) and the trace semantics ([38]). In the first case, two processes are equivalent iff they have associated the same transition trees after forgetting the states. In the second case, two processes are equivalent iff the corresponding sets of traces (streams of labels obtained travelling along all paths starting from the roots of the associated transition trees) are the same. In general, the semantics of processes depends on what we are interested to observe: i.e., the semantics of processes is observational.

In Fig. 5 we report the transition trees associated with two processes p_1 and p_2 which are equivalent w.r.t. the trace semantics but not w.r.t. the strong bisimulation.

One of the most interesting techniques for defining observational semantics for (finite and infinite) processes is the Park's bisimulation semantics (see [53, 51]). Assume that we have an lts $(STATE, LABEL, \rightarrow)$; then a binary relation R on $STATE$ is a *(strong) bisimulation relation* iff for all $(s_1, s_2) \in STATE$ s.t. $s_1 \, R \, s_2$

1. if $s_1 \xrightarrow{l} s'_1$, then there exists s'_2 s.t. $s_2 \xrightarrow{l} s'_2$ and $s'_1 \, R \, s'_2$
2. if $s_2 \xrightarrow{l} s'_2$, then there exists s'_1 s.t. $s_1 \xrightarrow{l} s'_1$ and $s'_1 \, R \, s'_2$.

Fig. 5. Transition trees associated with the processes p_1 and p_2

It can be shown that there exists a maximum bisimulation relation $\sim$ characterized by $\sim\, = \cup\{R \mid R$ is a bisimulation relation $\}$. What we get in this case is just the strong (bisimulation) equivalence; but there are many possible and interesting variations (see e.g. [51, 2]).

1.3 Formal Description and Specification

After choosing a particular kind of models for processes, there are two main approaches to formally describing (specifying) a process:

- *(model-oriented)* a process corresponds to a class of semantically equivalent models given by exhibiting a particular element of the class;
- *(property-oriented)* a process is specified by giving a set of properties, which determines a class of models (those having the required properties).

In general the above approaches have associated an appropriate syntactic support, respectively:

- a language s.t. each of its expressions corresponds to one model; then the language expressions may be used to define the processes via an appropriate semantic equivalence;
- a logical (specification) language s.t. its formulae express properties of the models together with a validity relation saying when a model has the property expressed by a formula; then a process is specified by giving a set of such formulae, which determine a class of models (those satisfying the formulae).

Algebraic specification of concurrency falls under the second (property-oriented) paradigm for which several logical languages have been proposed, based on modal / temporal logics in the case of lts's, event structures and Petri nets (see e.g. [56, 45]) and on first-order logic for characterizing dataflows.

In the literature several languages for supporting the first (model-oriented) approach have been proposed; among them we can recall CCS (Calculus of Communicating Processes) for lts's with strong bisimulation (see [50, 51]), CSP for lts's with trace semantics (see [38]), the various process algebras of the Amsterdam school (see e.g. [18, 14]) and so on.

The CCS approach has been recently expanded to a very interesting approach, the Π-calculus (and the mobile processes approach, see [52]), for dealing with processes which may exchange processes (identified indirectly by references) as messages. Notably the same problem has been first addressed and solved within an approach to the algebraic specification of concurrency, the SMoLCS approach (see e.g. [7] and Sect. 3.4) where processes are exchanged directly as values/data. This last viewpoint has been taken up and developed as an updating of CCS in [61].

In the following we briefly report, for example and further use, the full definitions of finite CCS and of the process algebra PA (see [14] Sect. 3).

CCS and process algebras represent the two prominent description styles based on the model of lts's. Milner defines inductively the transitions by SOS rules and then apart a semantics; while the Amsterdam group defines axiomatically the derived semantic equivalence (as initial semantics).

Finite CCS. Assume that $\mathcal{A}$ is a given set of basic action names and let $\overline{\mathcal{A}}$, $\mathcal{L}$ denote respectively the set of the complementary actions $\{\overline{a} \mid a \in \mathcal{A}\}$ and of the labels $\mathcal{A} \cup \overline{\mathcal{A}} \cup \{\tau\}$.

The set of CCS expressions $\mathcal{E}$ is inductively defined as follows.

- $nil \in \mathcal{E}$ (the process unable to perform any action)
- $\alpha.e \in \mathcal{E}$ for all $e \in \mathcal{E}$ and $\alpha \in \mathcal{L}$ (action prefixing)
- $e_1 + e_2 \in \mathcal{E}$ for all $e_1, e_2 \in \mathcal{E}$ (nondeterministic choice)
- $e_1 \parallel e_2 \in \mathcal{E}$ for all $e_1, e_2 \in \mathcal{E}$ (parallel composition)
- $e \backslash a \in \mathcal{E}$ for all $e \in \mathcal{E}$ and $a \in \mathcal{A}$ (restriction)

The lts associated with CCS is $(\mathcal{E}, \mathcal{L}, \rightarrow)$, where $\rightarrow$ is inductively defined by the following rules.

$$\frac{}{\alpha.e \xrightarrow{\alpha} e} \qquad \frac{e_1 \xrightarrow{\alpha} e_1'}{e_1 + e_2 \xrightarrow{\alpha} e_1'} \qquad \frac{e_2 \xrightarrow{\alpha} e_2'}{e_1 + e_2 \xrightarrow{\alpha} e_2'}$$

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{e_1 \parallel e_2 \xrightarrow{\alpha} e_1' \parallel e_2} \qquad \frac{e_2 \xrightarrow{\alpha} e_2'}{e_1 \parallel e_2 \xrightarrow{\alpha} e_1 \parallel e_2'}$$

$$\frac{e_1 \xrightarrow{\alpha_1} e_1' \quad e_2 \xrightarrow{\alpha_2} e_2'}{e_1 \parallel e_2 \xrightarrow{\tau} e_1' \parallel e_2'} \ \alpha_1 = \overline{\alpha_2} \text{ or } \alpha_2 = \overline{\alpha_1} \qquad \frac{e \xrightarrow{\alpha} e'}{e\backslash a \xrightarrow{\alpha} e'\backslash a} \ \alpha \neq a, \overline{a}$$

Two CCS expression are considered semantically equivalent iff they are strongly bisimilar (see Sect. 1.2). An equivalent alternative of Milner's CCS is the MEIJE Calculus of Boudol and Austry [13] with the related elegant foundational work of De Simone [31].

The process algebra PA. Let A be a given set of *atomic actions*; the set of the PA expressions E is inductively defined as follows.

- $\mathsf{A} \subseteq \mathsf{E}$ (processes performing just one atomic action)
- $x.y \in \mathsf{E}$ for all $x, y \in \mathsf{E}$ (sequential composition)
- $x + y \in \mathsf{E}$ for all $x, y \in \mathsf{E}$ (nondeterministic choice)
- $x \parallel y \in \mathsf{E}$ for all $x, y \in \mathsf{E}$ (parallel composition)
- $x \lfloor y \in \mathsf{E}$ for all $x, y \in \mathsf{E}$ (left merge)

Differently from the CCS case, here we do not associate an lts to the language (and then consider as semantically equivalent the expressions strongly bisimilar); we give instead directly a list of axioms identifying the expressions strongly bisimilar in the sense that equalities between terms in the initial model correspond to strong bisimulation equivalence. Notice that in order to do that we need to introduce an auxiliary operator ($\lfloor$).

$$x + y = y + x \qquad (x + y) + z = x + (y + z) \qquad x + x = x$$
$$(x + y).z = (x.z) + (y.z) \qquad (x.y).z = x.(y.z)$$
$$x \parallel y = (x \lfloor y) + (y \lfloor x)$$
$$a \lfloor x = a.x \qquad (a.x) \lfloor y = a.(x \parallel y) \qquad (x + y) \lfloor z = (x \lfloor z) + (y \lfloor z)$$

1.4 Illustrative Examples of Specification Problems

In order to give the flavor of what "specification of concurrent systems" means, we present some illustrative examples, which will be used in the sequel as reference for making more concrete some general considerations and for assessing different specification methods. Some very interesting specification problems (ten in all) have been proposed and discussed in a Cambridge Workshop, 1983, whose proceedings [32] may be of interest for the readers.

First we consider the problem of specifying a parameterized family of concurrent architectures: for each fixed set of parameters we get the specification of a family of essentially equivalent architectures. Then we briefly consider a more abstract level of specification, where we are concerned with looser requirements about architectures. The distinction here between the two cases looks rather fuzzy; however it has a great impact on the specification techniques. Later on when speaking of algebraic specifications we will see that this distinction will be similar to the one between non-loose and loose (or ultra-loose) algebraic specifications. In order to avoid confusion, in this paper we will call *abstract specifications* (AS) the first and *very abstract specifications* (VAS) the second ones; in the following section we will propose a more rigorous qualification of the two.

Specification of a Family of Concurrent Systems. The problem here is to specify a family of concurrent architectures (each one briefly called *CA*).

The structure of a *CA* is informally described below and graphically represented in Fig. 6, where the ovals represent the active components, the squares the passive ones and the straight lines the interactions among the components. A *CA* consists of a variable number of processes and a buffer shared among the processes; "variable" means that processes may terminate and new processes may be created. Processes can communicate among them by exchanging messages in a synchronous mode throughout channels (handshaking communication) and either reading or writing messages on the buffer; moreover the processes could also communicate with the outside world (consisting of other similar architectures) sending and receiving messages in a broadcasting mode; messages are simply values.

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

Fig. 6. A CA architecture

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

where x, ch, seq-c are respectively the nonterminals for variables, channel identifiers and sequential commands. A CA process may:

- write the value of a variable on the buffer, read a value from the buffer and assign it to a variable and test if the buffer is empty (the result of the test is assigned to a boolean variable) (1);
- exchange messages along the channels in a handshaking way and with the environment outside CA in a broadcasting way (2);
- create a new process with a given command part (and initial empty local memory) and terminate its execution (3);
- perform the sequential composition of two commands and nondeterministically choose between two commands (4);
- execute sequential commands not further specified (i.e. commands which do not require an interaction with other processes, nor with the buffer, nor with the environment outside CA) (5).

We are interested in specifying all instances of CA, where each instance is determined by an initial state, i.e., a set of processes and a buffer in some initial states.

To test the modularity and the possibility of giving specifications with "reusable" parts of the various methods we consider several versions of CA differing either for the kind of the data used by the processes, or the buffer organization, or the mode in which the process components perform their activity in parallel; each version of the architecture is denoted by $CA(D_i, B_h, A_j, P_k)$, where D_i, B_h, A_j, P_k represent respectively the assumptions on the data, the buffer organization, the buffer access and the kind of parallelism.

The data handled by CA processes may be: D_1) integer numbers, booleans and Pascal-like arrays of data; D_2) as D_1 but also the CA processes are data; D_3) as D_2 but also functions having as argument and result processes are data.

The buffer may be organized as: B_1) unbounded queue; B_2) unbounded stack; B_3) cell, which can contain at most one element.

The requirements on multiple simultaneous accesses to the buffer may be: A_1) several simultaneous accesses (i.e., writing, reading, testing) are not allowed; A_2) only several simultaneous testings are allowed; A_3) only several simultaneous testings and writings are allowed and the simultaneous writings of the values $v_1, \ldots, v_n$ could nondeterministically result into writing $v_1, \ldots, v_n$ in any order.

The requirements on the way the CA processes act in parallel may be: P_1) except for the synchronous interactions required by handshaking communications, the processes perform their activity in an interleaving mode; P_2) except for the synchronous interactions required by handshaking communications and the requirements on simultaneous buffer accesses, the processes perform their activity in a completely free parallel mode; P_3) as P_1 but reading takes precedence over writing.

The architecture CA_NET consists of several instances of CA in parallel, which interact by exchanging messages in a broadcasting mode (see Fig. 7). In this case the various components of the network perform their activity in a maximal parallel mode (i.e., no component can stay idle).

Fig. 7. CA_NET architecture

Abstract Requirements of a Computer Network. Here we want to express formally the abstract requirements for a computer network, without fully describing it. We need to express both requirements about the structure of the network (static properties) and about its activity (dynamic properties), as exemplified in the following

Static properties: the network must have between 10 and 20 computer components and must be able to store k bytes of information.

Dynamic properties: they may be further distinguished in:

safety properties: requirements about what should not happen during the system activity (e.g., no deadlock situation may arise [i.e., the network can stop its activity only when all computer components have terminated their activities], the percentage of the used storage will never be more than 95%);

liveness properties: requirements about what must happen during the system activity (e.g., if a command "run a certain correct C program" is given to one of the computers, then eventually the program result will appear on the computer screen; if a computer sends a message to another computer of the network, then eventually either the network breaks down or the message is received).

Note that we do not want to make any other assumption; thus we do not impose conditions about:

- the way the components are connected,
- the presence of auxiliary passive components (e.g., storage devices),
- the way the components interact among them and so on.

Thus to give the required specification means to formally identify the class of all networks satisfying the listed properties; this class includes, e.g., a network consisting of 10 Sun3 computers, five auxiliary disks connected by an Ethernet cable backbone; but also a network consisting of 15 PCIBM XT connected by a token ring.

2 Issues in Algebraic Specification of Concurrency

First we briefly review the aims of algebraic specifications of concurrency; then we try to identify the items/dimensions by which we can assess and relate different approaches.

2.1 Aims and Nature

Three Motivations/Viewpoints. While the meaning and the role of the formal specification of concurrent systems is clear, it is natural to ask why algebraic specifications, which have been invented for describing data structures and apparently deal with static data, should be adopted for describing intrinsically dynamic structures (and of course whether this approach is sensible and viable). There seem to be *three main motivations* for that, corresponding to *three views* of the relationship between data types and concurrent systems.

V1 Handling static adt's Concurrent systems use various data structures, whose abstract formal specification is most appropriately expressed by an algebraic specification.

V2 Abstract concurrent structure The abstract structure of a concurrent system (either globally or at a certain stage of its evolution) is conveniently described as an abstract data type.

V3 Abstract dynamic data types Concurrent systems themselves can be seen as data manipulated by other systems and functions, as any other data.

We will see those three motivations/views quite clearly reflected in various approaches, at a different level of integration.

We can now illustrate the different views by means of the examples of the *CA* architectures of Sect. 1.4.

V1 Under the assumption on the data handled by the processes D_1, for specifying a *CA* architecture we need to formally define a complex data structure (there are data of type integer, boolean and array with different dimensions). For giving an abstract specification of such architecture, we have to abstractly specify the above data structure and that could be conveniently done by an algebraic specification.

V2 Each *CA* architecture consists of an unordered group of processes plus a buffer shared among them, where the buffer organization depends on the assumption B_i. This structure may be abstractly defined by saying that an architecture is a couple consisting of a multiset of processes (since there may be two identical processes) and of a buffer, which is, depending on B_i, either a queue, or a stack, or a single value. Then it may be specified by giving first appropriate combinators for describing the buffer structure and for putting processes and buffer together, and later qualifying such operators by means of axioms. Thus the structure of a *CA* could be abstractly specified by an algebraic specification.

V3 We need a formalism fully supporting the third viewpoint whenever we want to specify the *CA* architectures following the assumptions on data D_2 (also *CA* processes are data) and D_3 (also functions from processes into processes are data) [2].

Methods and Instances. Whatever the motivation and the technical approach, a key distinction has to be made between "methods for" and "instances of" algebraic specification of concurrent systems; this is just the distinction between methods for specifying abstract data types and particular specifications of some data types. This remark is particularly relevant in concurrency, where algebraic techniques have found important applications in the description of abstract concurrent systems (viewpoint **V2**). For example, considering processes as data qualified by a set of axioms, we may obtain a theory of processes which is the analogous of a theory of rings or groups or, as a more familiar example to computer scientists, a theory of stacks or queues. Though this is a very important and fruitful viewpoint, we cannot consider these theories among the methods of algebraic specification of concurrency, which should instead provide techniques and guidelines for defining (classes of) abstract concurrent systems, as much as the classical algebraic specification formal methods for specifying (classes of) abstract data types. In particular it seems natural to require that methods of algebraic specification of concurrency reduce to some methods for the specification of abstract data types, i.e., that the viewpoint **V1** is always incorporated. For particular algebraic process theories, which are not methods, see, for example, the very elegant and informative book by M. Hennessy [37], where the axiomatizations of several simple concurrent languages are given.

A simple example of algebraic process theory, PA, has been given in Sect. 1.3. Here below we show how to turn the inductive definition of CCS given in the same section into an algebraic specification, which is an instance of the algebraic specification method based on conditional specifications (see [25] and Sect. 3.4).

[2] *CA* architectures following assumption D_3 are not unrealistic products of theoreticians but can be found in real systems: some Unix commands are modelled as functions from processes into processes and the denotation of an Ada task type is a function returning a process.

spec CCS =

sorts exp, act, lab

opns

$a_1, a_2, a_3, \ldots : \rightarrow act$

$-: act \rightarrow lab$

$\overline{-}: act \rightarrow lab$

$\tau: \rightarrow lab$

$nil: \rightarrow exp$

$-.-: lab \times exp \rightarrow exp$

$- + -: exp \times exp \rightarrow exp$

$- \parallel -: exp \times exp \rightarrow exp$

$-\backslash -: exp \times act \rightarrow exp$

preds

$- \xrightarrow{-} -: exp \times lab \times exp$

axioms

$\alpha.e \xrightarrow{\alpha} e$

$e_1 \xrightarrow{\alpha} e_1' \supset e_1 + e_2 \xrightarrow{\alpha} e_1' \qquad e_2 \xrightarrow{\alpha} e_2' \supset e_1 + e_2 \xrightarrow{\alpha} e_2'$

$e_1 \xrightarrow{\alpha} e_1' \supset e_1 \parallel e_2 \xrightarrow{\alpha} e_1' \parallel e_2 \qquad e_2 \xrightarrow{\alpha} e_2' \supset e_1 \parallel e_2 \xrightarrow{\alpha} e_1 \parallel e_2'$

$e_1 \xrightarrow{t} e_1' \wedge e_2 \xrightarrow{t'} e_2' \supset e_1 \parallel e_2 \xrightarrow{\tau} e_1' \parallel e_2'$ for all t, t' s.t. $t' = \overline{t}$ or $t = \overline{t'}$

$e \xrightarrow{t} e' \supset e\backslash a \xrightarrow{t} e'\backslash a$ for all $t \neq a, \overline{a}$

The initial model I_{CCS} of CCS determines an abstract lts, whose sets of states and labels are respectively $(I_{CCS})_{exp}$, $(I_{CCS})_{lab}$ and whose transition relation is $\rightarrow^{I_{CCS}}$; then the elements of $(I_{CCS})_{exp}$ are further identified by bisimulation semantics. Notice that, assuming as primitive the sorts act and lab, the final semantics of CCS as a hierarchical specification coincides with the initial semantics.

2.2 Items for Taxonomy

Guided by the discussion about the aims of Sect. 2.1, we try to define some items for locating various approaches and obtaining a reasonable taxonomy. We will refer implicitly to those items in the following when discussing some approaches. We do not pretend all the items to be completely independent, and this is why we do not call these items just dimensions; anyway their relationship should be clear from our presentation.

Specification Formalism. Even when it is not explicitly stated and whenever possible, we try to identify a *basic specification formalism* more or less under the institution paradigm, in the sense of Burstall and Goguen [26].

A *basic specification* consists of a set of sentences over a signature; its models are those structures on the signature satisfying the sentences. Sometimes (basic) specifications denote classes of structures satisfying some properties; we will stick to the syntactic view of a specification as a presentation, so that it makes sense to speak of semantics of specifications.

It is important here to distinguish the models of a specification from what we call *basic concurrent models*; i.e., the structures that represent semantically processes/concurrent systems (see Sect. 1.1).

Also, as we anticipated in Sect. 2, we would normally expect that an algebraic specification formalism for concurrency reduces to an associated *algebraic specification formalism for abstract data types*, whenever one does not consider processes, but just static data types.

Starting with a *basic specification formalism*, we may build more complex specifications by suitable operations over basic specifications. A linguistic support to such operations constitutes a *specification language*.

Support for Concurrency. There are two aspects by which concurrency is dealt with in a method for the formal specification of concurrency: the *basic concurrent models* for processes and the *primitives for concurrency*.

Typical *basic concurrent models* are labelled transition systems with the associated, possibly infinite, labelled transition trees, Petri nets of various kinds, stream processing functions (dataflows), which have been briefly illustrated in Sect. 1.1. There are also approaches, like process algebra, apparently qualifying processes just by axioms without any reference to an underlying structure. But inevitably any such axiomatization is driven by a hidden structure and any method which does not explicitly refer to a model, either provides a manner of descriptions of basic concurrent models or leaves the burden of identifying and describing such a model to the user.

By *primitives for concurrency* we refer to higher-level aspects of concurrent systems concerning operations for structuring processes, and the mode of the interactions of processes and of their evolutions. Typical primitives are: operations for composing processes in parallel or hierarchically; communication mechanisms like synchronous and asynchronous message passing and shared variables; global evolution modes, like interleaving, free and maximal parallelism.

For specifying the *CA* architectures of Sect. 1.4, for example, we need to consider processes interacting by synchronously exchanging messages and by accessing a shared buffer; while the nodes of the network *CA_NET* interact by broadcasting communication; moreover such processes may evolve in parallel either in an interleaving or free mode (P_1 and P_2) or in an interleaving mode where the conflicts on the buffer access are solved by giving precedence to reading w.r.t. writing (P_3).

Semantics and Level of Abstraction. As for classical adt specifications, here specifications may determine one or more abstract concurrent systems, where by abstract concurrent systems we mean an isomorphism class of concurrent systems; those abstract concurrent systems are the *semantics* of the specification.

In the first case it is intended that a specification determines just one isomorphism class. In the second case the semantics (sometimes we just say the specification) is *loose*; in the classical sense a loose semantics determines a class of non-isomorphic structures on Σ; we may also speak of *ultra-loose* specifications in various senses, for example when we consider structures also over signatures with some relation with Σ. Since there is not a general agreement on the meaning of loose and especially of ultra-loose, in the present context we prefer to speak of *very abstract specifications (VAS)*, qualifying instead as *abstract specifications (AS)* those identifying one abstract concurrent system.

Semantics is one of the most delicate and interesting topics in concurrency; we will see that essentially three approaches are followed:

- adopting a given semantics for processes, just connecting it to a semantics for usual adt's; this is possible whenever there is a clear separation between processes and the data they manipulate;
- defining, by adding suitable axioms, a classical semantics, e.g., initial or final, in a way that captures the wanted semantics for processes;
- providing new semantic paradigms for adt's, which are able to accommodate semantics sensible for data which are processes, thus extending the classical semantic theory for adt's.

We will discuss in some more detail the issue of semantics in the following section.

It turns out that VAS of concurrent systems need specification formalisms (say institutions) and semantics which go beyond those in use for classical static adt's in two respects. Notions of ultra-loose semantics have to be developed in order to abstract from the particular structure of an architecture; these abstractions cannot be captured simply by means of the classical notion of satisfaction. Moreover we have to go beyond equational and conditional logic, since, in order to express requirements on events and their relationship (sometimes distinguished in safety and liveness properties) we need the power of first-order logic, possibly with infinitary conjunctions, and of various forms of temporal logic. This necessity of encompassing classical adt specification for VAS explains why most algebraic methods are mainly concerned with AS. Thus we will have to deal mainly with AS and only briefly we will discuss the VAS issue.

Integration. By integration we refer to the two following distinct aspects:

- the level of integration of abstract data types and concurrent features;
- the extent to which a specification of a concurrent system is truly an algebraic specification and can exploit classical concepts and results of general algebraic specifications.

For classifying the integration of adt's and concurrent features we may refer to the three viewpoints illustrated in Sect. 2 and related to three motivations for making specification of concurrency algebraic: *inclusion of static data types* (**V1**) in a fixed concurrent structure, *algebraic specification* also *for the concurrent structure* (**V2**) a finally considering processes as data themselves, what we call *dynamic data types* (**V3**). As long as a method always subsumes as a special case a general method for specifying classical adt's, we may consider the three above viewpoints arranged in an order of increasing integration. Clearly the dynamic data type viewpoint corresponds also to an overall algebraic framework, but different possibilities arise, as we shall see, especially following the first viewpoint, which leaves separate the specification of the static adt's and of the concurrent structure, but may result in a completely algebraic specification.

Applications. It seems that we are still at an infancy stage for speaking in general of applications to real industrial cases. However it makes sense to look at applications in a broader sense: use of a method in proposal for standard tools, in research projects, in prototyping, in industrial case-studies. To some extent we may also guess at potential applications, looking at the varieties of example applications a method has shown.

Tools. It is a widely accepted dogma that there is no hope of real use of algebraic specifications without a convenient toolset for editing, verifying and animating specifications. Specification of concurrency adds a new challenge in many senses: concurrent rewriting systems are typically non-confluent and non-terminating; even when applicable, some general tools for algebraic specifications are inefficient to master the complexity of the dynamic behaviour of processes; finally the verification procedures, in order to deal with various observational semantics for processes, need specific techniques, far beyond the case of equational or conditional deduction.

Pragmatics. Note that the point here is not just feasibility or expressive power, but also convenience. As M. Broy points out in [22]

> "most important properties of specification methods are not only the underlying theoretical concepts but more pragmatic issues such as readability, tractability, support for structuring, possibilities of visual aids and machine support".

2.3 Beyond Classical Algebraic Specifications

The algebraic approach to specification applied to concurrency poses some new problems, whose solution is of general interest to the theory of algebraic specifications. A typical issue which has stimulated new techniques, beyond the classical ones, is semantics. We present briefly two recent theories, which are representative of the two main approaches to the algebraic semantics of processes. The first, "projection specifications", is an interesting variation of the initial approach taken in the process algebra school; in this approach the axioms embed a particular semantics and processes are limit points in a complete metric space, which is also a continuous initial algebra.

The second, "observational specifications", takes the bisimulation approach (typical of CCS) extending it to general algebraic specifications; it encompasses, as specializations, most presently known sensible semantics for processes together with all classical semantics for adt's.

Projection Specifications When specifying processes frequently we need to consider "infinite elements"; for example, when processes are modelled by means of (possibly) infinite trees, when the result or the observation of a process is a (possibly) infinite stream of values and so on. The projection specifications developed by the Ehrig's group in Berlin, and especially by M. Grosse-Rhode in his Diplom thesis (see [35, 36]) allows us to specify algebraically infinite objects.

The key ideas of projection specifications are the following:

- infinite objects are seen as limit points in complete metrics spaces, called projection spaces, since the metric is defined in terms of projections; moreover the continuity of operations ensures that the models are continuous algebras;
- the algebraic specifications of infinite objects include the specification of projections and thus embody the metric; the associated continuous model is obtained by a standard construction.

This approach is much related to the metric approach developed by the de Bakker's school in Amsterdam (see [30]) as a development of a pioneering paper by Arnold and Nivat [1]. The key difference here is the explicit specification of projections.

Projection specifications are almost as usual total equational algebraic specifications, but with an explicit projection operation for each sort $p\text{-}s\colon nat \times s \to s$; some constraints are added thus obtaining the constrained projection specifications to ensure conservative extensions of the naturals, if used in the projections, and compatibility of operations with projections.

Projection algebras are models of constrained projection specifications; they are characterized by the fact that every carrier is a projection space and the operations are projection compatible. By defining suitable projection morphisms, for every projection specification PS, the projection algebras which are PS models become a category $Cat(PS)$, which admits (free and) initial algebra T_{PS}.

By a standard construction procedure on projections, with every projection algebra a continuous algebra is associated which is complete (i.e., every carrier is a complete projection space) and separated (i.e., equality of projections implies equality of the limits). Thus the semantics of a projection specification PS is the initial complete separated algebra CT_{PS} which is associated with the initial algebra T_{PS} and is initial in the category of complete separated projection algebras.

Moreover projection specifications have nice properties like existence of free construction and the amalgamation and extension properties, so that the modularization and parameterization techniques and the results for classical equational specifications may be fully extended to projection specifications.

The theory of projection spaces and specifications provides a way of specifying processes as infinite objects of some kind, defining their projections, The nice aspect is that whenever processes are specified in this way, there is a full integration of data and processes, because processes are data themselves (viewpoint **V3**: dynamic data types). Moreover this theory, not being bound to a particular model of processes, nor languages, nor method, can be incorporated in other methods (for example, it seems possible to use it within the SMoLCS approach, see Sect. 3.4, for associating semantics in a pure algebraic way as in [6]).

Here we report the projection specification of the process algebra PA already considered in Sect. 1.3, but in this case we have the initial complete separated algebra which considers also infinite processes, while in 1.3 only the finite ones are considered.

spec $PA\text{-}Pr$ =
  enrich $PNAT$ **by**
  sorts exp
  opns
    $a_1, a_2, a_3, \ldots \colon \to exp$
    $-.-\colon exp \times exp \to exp$
    $- + -\colon exp \times exp \to exp$
    $- \parallel -\colon exp \times exp \to exp$
    $-\lfloor -\colon exp \times exp \to exp$
    $p\text{-}exp\colon nat \times exp \to exp$
  axioms
    $x + y = y + x \qquad (x + y) + z = x + (y + z) \qquad x + x = x$

$(x+y).z = (x.z) + (y.z) \qquad (x.y).z = x.(y.z)$
$x \parallel y = (x \lfloor y) + (y \lfloor x)$
$a \lfloor x = a.x \qquad (a.x) \lfloor y = a.(x \parallel y) \qquad (x+y) \lfloor z = (x \lfloor z) + (y \lfloor z)$
for all $i \geq 1$
$\quad p\text{-}exp(n, a_i) = a_i$
$\quad p\text{-}exp(0, a_i.x) = a_i$
$\quad p\text{-}exp(Succ(n), a_i.x) = a_i.p\text{-}exp(n, x)$
$p\text{-}exp(n, x+y) = p\text{-}exp(n, x) + p\text{-}exp(n, y)$

Here $PNAT$ is the projection specification of natural numbers with the usual operations 0 and $Succ$.

Observational Structures and Specifications The usual semantics for basic algebraic specifications in general are not adequate for specifications of processes. Consider, for example, the algebraic specification CCS of finite CCS given in Sect. 2.1. Initial and final semantics of CCS, which coincide, do not represent the right semantics for CCS (for example, nil and $nil + nil$ are not identified, since there are no axioms requiring the identification of terms of sort exp).

Also the common form of observational semantics for algebraic specification (sometime called behavioural semantics as in [66]) allows only to express particular bisimulation semantics: those which may be characterized by primitive observations (see [10] for a general treatment and [6] for the particular case of weak bisimulation; while in [12] a lattice of simulation relations is defined, whose greatest element can be seen as a possible correspondent of bisimulation in an algebraic framework). However behavioural semantics cannot be used for general bisimulation semantics, for example in the case of higher-order processes (i.e., when processes are communicated among processes).

The semantics of CCS is given usually by means of several variations of bisimulation. However various problems are encountered when extending bisimulation semantics to algebraic specifications. Consider the higher-order extension of the finite CCS (algebraically specified in Sect. 2.1) which is formally defined below.

spec $HCCS$ =
enrich CCS **by**
opns
$Pcom: exp \rightarrow act$
– also communicating a process is a basic action

In this case a bisimulation semantics must determine an algebra; thus instead of bisimulation relations we have to consider families of relations on the carriers of the initial model indexed on the sorts of the specification $R = \{R_s\}_{s \in \{exp, act, lab\}}$; so if the maximum bisimulation is a congruence, then the resulting semantics will be the quotient algebra I_{HCCS}/R.

Then we have to give conditions for R being a bisimulation, which must differ from those of 2.3, otherwise $Pcom(nil).nil$ and $Pcom(nil + nil).nil$ will be distinguished by the maximum strong bisimulation (while each reasonable extension of strong bisimulation should identify them, since both communicate strongly bisimilar processes); more generally the condition for R_{exp} being a bisimulation of 1.2 can be rephrased as follows. For all e_1, e_2 s.t. $e_1 \, R_{exp} \, e_2$

- if e_1 passes an experiment $EXP_1 = x \xrightarrow{l_1} e'_1$ (i.e. the formula $x \xrightarrow{l_1} e'_1$ holds on e_1), then e_2 passes another experiment $EXP_2 = x \xrightarrow{l_2} e'_2$ R-similar to EXP_1;
- analogous condition for e_2;

where $x \xrightarrow{l_1} e'_1$ R-similar to $x \xrightarrow{l_2} e'_2$ means $l_1 = l_2$ and $e'_1 \, R_{exp} \, e'_2$.

For coping with higher-order CCS we have just to modify the notion of R-similar experiment; now $x \xrightarrow{l_1} e'_1$ R-similar to $x \xrightarrow{l_2} e'_2$ iff $e_1 \, R_{exp} \, e_2$ and either $(l_1 = l_2)$ or $(l_1 = Pcom(e'),\, l_2 = Pcom(e'')$ and $e' \, R_{exp} \, e'')$ or $(l_1 = \overline{Pcom(e')},\, l_2 = \overline{Pcom(e'')}$ and $e' \, R_{exp} \, e'')$.

What about R_{act} and R_{lab}? We have to extend the identifications on processes to the elements of other sorts, since they may have process components.

R_{act} : For all a_1, a_2, $a_1 \, R_{act} \, a_2$ implies
either $(a_1 = a_2)$ or $(a_1 = Pcom(e'),\, a_2 = Pcom(e'')$ and $e' \, R_{exp} \, e'')$.
R_{lab} : For all l_1, l_2, $l_1 \, R_{act} \, l_2$ implies
either $(l_1 = a_1,\, l_2 = a_2)$ or $(l_1 = \overline{a_1},\, l_2 = \overline{a_2})$ with $a_1 \, R_{act} \, a_2$.

The above problems (and many others) are tackled within the theory of observational structures and specifications, which we sketch below (see [2] for a full treatment). The result is a new theory of adt semantics on its own, encompassing the classical treatment.

An *observational structure* essentially consists of a first-order structure (or algebra with predicates) equipped with

- *experiments*: (possibly infinitary) first-order contexts for observable elements;
- a *similarity law* for experiments: a function which, given a (similarity) relation on the elements of the algebra, generates a similarity relation on experiments;
- a *propagation law* for relations: a function which propagates a (similarity) relation on the observable elements to a (similarity) relation on elements of the other sorts.

With each observational structure a family of *observational relations* is associated, with a maximum that we call *observational equivalence*. This equivalence, as expected, is not always a congruence; thus it is shown how to derive canonically an approximating congruence and also how to define observational equivalences which are congruences. Whenever this equivalence is a congruence we get an observational semantics by the usual quotient operation.

The construction is a much abstract version of Park's construction of maximum bisimulation (see Sect. 1.2). Indeed, observational structures capture the essential ingredients for defining over algebraic structures those semantics which share with the original notion of bisimulation semantics the feature of being maximum fixpoints of suitable transformations. Hence the associated proof technique is effective: in order to show that two elements are semantically equivalent, just find an observational relation to which they belong. As a desired consequence many known bisimulation semantics for processes (presently, all known to us) are special cases of this construction. But *observational structures are not at all confined to a generalization of bisimulation semantics for processes.* Indeed because of their abstract nature and of the flexibility in the choice of the similarity laws for experiments and of the propagation laws for relations, they can be applied to give a wide range of semantics for abstract data types. It can be shown indeed that the full class of well-known semantics, like initial, final and various behavioural semantics, are special cases of this paradigm.

An *observational specification* is a particular case of observational structure in which we make explicit use of an algebraic specification *SPEC*; moreover the algebra component of the structure is the initial model of *SPEC* and the propagation and similarity laws are derived by the axioms of the specification as follows.

- The *free axiomatic propagation law* provides the minimal propagation of the identifications on the observed elements to the whole structure (i.e., to all nonobserved sorts and all predicates) which preserves the algebraic structure and the validity of the specification axioms about non observed elements and predicates.
- The *free axiomatic similarity law* considers equivalent two experiments if and only if they at most differ for subcomponents which are related by the above free axiomatic propagation law.

3 Outline of Some Approaches

We have selected a number of relevant approaches, which are briefly presented with an implicit reference to the items of Sect. 2; moreover for each one we comment its application to the example specification problems (the *CA* architectures) presented in Sect. 1.4.

3.1 The Process Specification Formalism for Process Algebras (PSF)

PSF (see [46]) is the process specification formalism developed by Mauw and Veltink as a base for a set of tools to support process algebra of Bergstra, Klop et al' (see e.g., [18, 14]). The main goal in the design of PSF was to provide a specification language with a formal syntax similar to the process algebra ACP (see [14] Sect. 4) but also with a notion of data type; to this end ASF (the Algebraic Specification Formalism of [17], which is based on the formal theory of abstract data types) has been incorporated.

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

A PSF specification consists of a series of modules, distinguished in data modules and process modules.

Data modules are algebraic specifications of adt's with initial semantics.

Process modules are algebraic specifications of processes. Formally a process module has the following form.

process module NAME
  begin
    atoms – atomic action symbols declaration
      $A\colon s_1 \times \ldots \times s_n$

...

processes – process symbols declaration

$P: s_1 \times \ldots \times s_n$

...

set of atoms – set of atoms used with the hiding and encapsulation operation declaration

$H = \ldots$ – using set comprehension

...

communications – explicit definition of synchronous actions and of the resulting label

$a_1 \mid a_2 = a_3$

...

definitions

$P(x_1, \ldots, x_n) =$ ACP-expression

end NAME

Processes are particular data structures obtained from operators like "+", "||", ";", "**hide**" and "**encaps**", elementary processes called atomic actions and recursive definitions; the given (equational) axiomatization determines a particular semantics over these structures embodying ideas of concurrency. This is best understood looking at the hidden basic concurrent model behind process algebra which are lts's as in CCS and many other approaches; then the axioms provides semantics like strong, trace or bisimulation semantics and others. The hidden model is made evident in some presentations of PSF, where ACP processes are described by means of SOS-like rules (see [54]) describing transitions. Anyway, since ACP provides essentially a language schema for processes, it is irrelevant, except for building the tools, how its semantics is given, either by equations or by transitions plus semantic equivalences.

It is instead important to notice that in PSF:

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

The interface between processes and adt's is as follows:

- the atomic actions may have as components values of the specified adt's;
- it is possible to define recursively families of processes indexed on the elements of some sort;
- an infinitary non-deterministic choice indexed on the elements of some sort is available.

Note that there is no notion of global data and the communication mechanism is by message passing.

The semantics of the data part is a classical algebraic semantics by initiality; the semantics of processes is in general a bisimulation semantics which gives a congruence on the term algebra. Thus the semantics identifies an isomorphism class of structures, i.e., an adt.

The data part is strictly distinguished from the process part, i.e., the first viewpoint of including the specification of static adt's into a formalism for concurrency is followed; but also the concurrent structure is here specified algebraically though with a fixed set of primitives parameterized on the actions and on synchronization structure. The result is a completely algebraic specification to which all the techniques and results of ASF can be applied nicely.

Particularly powerful are the modularization mechanisms in PSF, which are borrowed from ASF but are truly dealing with integration of adt's and processes: the module concept supports importing and exporting also of processes and actions.

There is a vast literature on the use of process algebra with a detailed treatment of classical examples and correctness proof for implementation. However these examples should not be confused with applications of a specification method like PSF, which has been introduced indeed for supporting industrial applications. Clearly PSF is practically applicable to a wide range of significant cases, 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. [46]). This toolset is currently in an advanced phase of development ([47]); in particular a simulator, a term rewriting and a proof assistant has been implemented. From the design of the toolset, it seems that this would be a most interesting feature of PSF.

Some of the CA architectures may reasonable specified using PSF; clearly the buffer has to be realized by means of a particular process (all buffer organizations may be handled).

The following variations of CA cannot be specified in PSF: D_2 and D_3 (PSF does not support the viewpoint **V3**) and also P_2 and P_3 since PSF allows only interleaving execution mode for processes and does not offer a way to solve conflicts; for the same reason also CA_NET cannot be specified in PSF.

3.2 LOTOS

LOTOS has been probably the first internationally known (since 1984) algebraic specification formalism for concurrency (see [27, 40]); most importantly it is an official ISO language specification 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 to PSF for understanding its structure.

LOTOS adds classical adt specifications into a language for concurrency as PSF; but it uses ACT ONE ([34]) instead of ASF and a process description based on an extension of CCS with several derived combinators (e.g. input/output of strucutured 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) is the same and also bisimulation semantics for processes.

PSF is an improvement over LOTOS (see a discussion in [47]), since it allows more freedom in the definition of synchronization mechanisms and supports import/export of action/processes thus being more flexibile for stepwise development.

Along all these years LOTOS has been used in several practical applications and moreover nowadays a toolset for helping to write down LOTOS specifications has been developed (see e.g. the ESPRIT project LOTOSHERE [62]).

3.3 ACP with Shared Data

This approach is due to S. Kaplan and has been presented in [42, 43]. The situation is very similar to the one in PSF: strict hierarchical separation between data and processes and processes specified by a schematic language. The difference is that now the actions operate on the data and thus what is parametric is now the specification of the effect of actions on data.

The basic specification formalism is equational logic with total algebras. Any specification language that supports this basic formalism can be used to provide methods for modularization and parameterization; PLUSS (see [21]) is explicitly quoted as such a possible language.

A process specification is in two layers:

- the data part, which is a classical algebraic specification of some basic data structures; data represent the concurrent states of the (global) environment which is manipulated by processes;
- the *process* part, specifying the agents that act concurrently on the data. Processes are built out of basic entities called *actions*, by means of the operators (those typical of process algebra) ";", "||", "+".

The interactions of processes with data is modelled by an application operator $- :: - : process \times data \to data$. For example we can build a process stack by means of operations like $PUSH\colon item \to process$; then we can specify its effect with an axiom like: $PUSH(i) :: d = Push(i, d)$.

The states of a concurrent system have always the form "*process* :: *data*" and they are of type data; for example

$$(PUSH(i_1) \mid\mid PUSH(i_2) \mid\mid PUSH(i_3) \mid\mid POP) :: Push(i, Empty).$$

For making easier the specification, it is possible to declare composed actions, i.e., processes built from atomic actions by ";", "+" and "||", also in a parameterized way. The general form of a process specification is as follows:

process specification: *SPEC*
  data specification: *DATA*
  atomic actions: *A*
  composite actions: *C*
  equations for atomic actions: EQ_A
  equations for composite actions: EQ_C
end process specification: *SPEC*

The communication mechanism is by shared data and indeed the actions are used for manipulating the global data, while there is no provision for message passing.

The basic concurrent model is the process algebra ACP: i.e., no explicit model is given since processes are treated as static data, but the hidden models are lts's. The axioms for processes axiomatize the execution mode by interleaving of atomic actions and trace semantics. Note that trace semantics refers to the basic model for concurrency and not to the overall semantics for processes (see below).

Semantics is done by purely algebraic methods, i.e. it is hierarchical on data specification with an overall observational semantics, given by a congruence corresponding to a final algebra. Indeed the semantics of a process specification *SPEC* is given by translating $SPEC$ into a classical algebraic specification $SEM(SPEC)$. The specification $SEM(SPEC)$ has a standard part consisting in equations for defining the operators on processes (those of ACP) and the semantics of the application operators of processes to data.

The overall semantics of $SEM(SPEC)$ is observational in an algebraic sense, i.e. it gives an observational congruence corresponding to a final algebra; this semantics formalizes an input-output semantics for processes with respect to data, as it is sensible for a concurrent system which represents non-deterministic transitions from a data configuration to another one (recall that the overall state of the system is of sort data).

Infinite processes are possible by means of fixpoint operator "$\mu\, p.E[p]$", whose semantics is axiomatically given by the unfolding equation as usual; dynamic processes, which may modify their structure depending on the environment, are obtained by a conditional operator.

Being fully algebraic, the approach can use various operators for composing specifications and of course parameterization and notions of implementation (and indeed the approach was developed in the same environment of PLUSS and ASSPEGIQUE). But it also provides a special composition operator for process specifications "$|||$" showing that under certain conditions one can convert a specification $SPEC_1 \,|||\, \ldots \,|||\, SPEC_n$ into a basic specification. This operation is viewed as a first step towards multilevel structuring, which is not exploited.

There is no mention in the literature of significant applications; however it can be easily understood that the method is viable whenever and only when we have to model concurrent architectures where the communications is by shared data. Converting models with message passing would imply some coding, which inevitably affects the level of abstraction.

No special tools. If the processes are finite then it is understandable that the usual tools for algebraic specification work.

Some of the CA architectures may be specified using this approach; however the handshaking communication between processes has to be simulated using particular shared structures and so, as said before, what we get is more an implementation of the architecture that an abstract specification. Clearly all kinds of buffer organization (variations B_1, B_2 and B_3) may be specified. Moreover, since also this formalism is based on the ACP, it has has the same restrictions of PSF: so we cannot specify neither the CA architectures corresponding to variations D_2, D_3, P_2, P_3 nor the network CA_NET.

3.4 Dynamic Specifications and SMoLCS

The core of the SMoLCS approach has been developed, mainly by E. Astesiano and G. Reggio at the University of Genova, with a significant contribution by M. Wirsing of the University of Passau, since 1983, first within the Italian national project CNET (Campus Net, the prototype design of a local area network). While the core of the method has been unchanged, significant improvements and additions have been made since, especially for what concern semantics, tools and specifications at higher-level of abstraction. Curiously enough its theoretical development has been always accompanied and partly driven by applications; indeed at the time of its appearance in the international literature [3, 4] a full SMoLCS specification of the prototype CNET communication architecture had been already completed [4]; the first tools have been developed for application to the draft formal specification of full Ada; the most recent development for very abstract specifications have been required for applications to two industrial case studies.

As a most distinctive feature, SMoLCS supports, within one specification formalism, different ways of specifying concurrent systems, adapting the description to the level of abstraction of the specified system. In its current version it even supports explicitly various forms of very abstract specification, which will be discussed among others in a later section; here we present just abstract specifications.

Any institution which support conditional specifications with predicates can be used as algebraic specification formalism; a privileged one is $\mathcal{CONDYN}$, the institution of conditional dynamic specifications with partial (total) [order-sorted] algebras with predicates. Though most examples have been given in an ASL-like metalanguage [11], SMoLCS is not bound to any particular specification language, as long as it supports at least conditional specifications with predicates, parameterization and modularization mechanisms.

SMoLCS is centered around the following ideas.

- Processes are specified algebraically as lts's and are themselves data as any other; thus can be manipulated by functions and processes (viewpoint **V3**: dynamic data types); in particular higher-order concurrent systems and calculi are supported (actually they have been first introduced and developed within the SMoLCS approach, see [7]).
- It supports the user-defined specification of any kind of concurrent structure, communication mechanism (from message passing to shared data) and execution modes (from interleaving to priorities).
- This support is provided by modularization, hierarchization and parameterization mechanisms for defining and combining parts of a system, with possibly reusable components of any kind (data, actions, communication and execution mode schemas).
- Semantics also is user defined, following a schema for defining observations of the system depending on a viewpoint.

A process is specified by giving a dynamic specification (an algebraic transition system, as it was called until '89), which is as follows.

- A *dynamic signature* $D\Sigma$ is a couple (Σ, DS) where:
  * $\Sigma = (S, OP, PR)$ is a predicate signature,
  * $DS \subseteq S$ (the elements in DS are the *dynamic sorts*, i.e. the sorts corresponding to states of lts's),
  * for all $st \in DS$ there exist a sort $l\text{-}st \in S - DS$ (the sort of the labels) and a predicate $- \xrightarrow{-} -: st \times l\text{-}st \times st \in PR$ (the transition predicate).
- A *dynamic algebra* on $D\Sigma$ (shortly $D\Sigma$-algebra) is just a Σ-algebra.
- An *absract dynamic data type* (shortly *addt*) is an isomorphism class of dynamic algebras on a signature.
- A couple $(D\Sigma, Ax)$, where $D\Sigma$ is a dynamic signature and Ax a set of first-order formulae on $D\Sigma$ is called a *dynamic specification*.

The axioms may refer both to static aspects (e.g., values, states of a system) and to the dynamic aspects, i.e. concerning the transitions predicates.

By algebra we mean usually a many-sorted algebra with predicates; though we generally prefer, for reasons of convenience in applications, to use partial algebras, there are no problem to use, for example, total or order-sorted algebras.

There are various institutions of dynamic specifications depending on the form of the axioms. For the purpose of abstract specifications the most interesting is $\mathcal{CONDYN}$, the institution of conditional dynamic specifications, with axioms of the form

$$\wedge_{i=1,\ldots,n}\alpha_i \supset \alpha,$$

where α_i and α are atoms, i.e., formulae of the form $t = t'$ or $Pr(t_1, \ldots, t_n)$ with Pr predicate symbol (in the case of partial algebras the equality in the formulae is interpreted as existential equality).

$\mathcal{CONDYN}$ has some nice features; indeed a conditional dynamic specification has always an initial model which defines an associated lts; moreover it is possible to make a clear distinction between static and dynamic axioms, the last ones being those where α has the form $s \xrightarrow{l} s'$.

Depending on the degree of separation between static and dynamic aspects in the axioms, various simple inductive ways of defining the associated lts's are possible.

- A dynamic specification determines an addt, according to a semantics that can be user defined following a schema for defining observational semantics. There are essentially two ways in SMoLCS for associating a semantics: by adding axioms defining observations and thus getting semantics as a terminal semantics (a terminal congruence, see especially [6]), or by defining over the specification an observational structure, as specified in [2] (see also Sect. 2.3) and getting an observational equivalence, which has to be proven to be a congruence; sufficient conditions are given to ensure this.

  Observational equivalence includes as a special case all presently known sensible semantics for concurrency like trace and bisimulation (strong, weak, branching, distributed), etc. Since observational semantics is obtained as a maximum fixed point of a suitable monotonic transformation, the same proof technique of bisimulation can be applied.

- A support to modular specification of concurrent systems is then given accordingly to the following schema, where we outline the methodological aspects, leaving apart the algebraic formalism, which can be found in the quoted papers.

  A *concurrent system* is specified as follows: the states $(p_1 \mid \ldots \mid p_n \mid i)$ are a multiset of states of the process components and a value i representing the global information.

  The transitions are specified splitting the specification in several steps, where at each step some partial moves are defined using the partial moves defined at the previous step; at the first step the partial moves are defined starting from the transitions of process components.

The rules defining the transitions at the first step have form

$$\frac{p_i \overset{pl_i}{\Longrightarrow} p_i' \qquad i = 1, \ldots, n}{p_1 \mid \ldots \mid p_n \mid mp \mid i \overset{l}{\longrightarrow} p_1' \mid \ldots \mid p_n' \mid mp \mid i'} \quad cond$$

where $\Rightarrow$ denotes the transition relation of the process components, mp, i, i', the pl_1's, l and $cond$ are metaexpressions and the p_i's and p_i''s are metavariables not appearing in $cond$; while the rules defining the transitions of the next steps have form

$$\frac{mp_j \mid i \overset{l_j}{\sim\!\sim\!\sim\!\sim\!\sim\!>} mp_j' \mid i_j' \qquad j = 1, \ldots, n}{mp_1 \mid \ldots \mid mp_n \mid mp \mid i \overset{l}{\longrightarrow} mp_1' \mid \ldots \mid mp_n' \mid mp \mid i'} \quad cond$$

where $\sim\!\sim\!\sim\!\sim\!\sim\!>$ denotes either the transition relation corresponding to the partial moves of the previous steps or $\rightarrow$, mp, i, i', the l_j's, i_j', l and $cond$ are metaexpressions and the mp_j's and mp_j''s are metavariables not appearing in $cond$.
It is shown that any specification of a concurrent system may be reduced to a canonical form consisting of the composition of three particular steps, respectively for synchronization, parallelism and monitoring, which are characterized by rules of the the form respectively:

Synchronization
$$\frac{p_j \overset{l_j}{\Longrightarrow} p_j' \qquad j = 1, \ldots, n}{p_1 \mid \ldots \mid p_n \mid i \overset{sl}{\longrightarrow} p_1' \mid \ldots \mid p_n' \mid i'} \quad cond$$

Parallelism
$$\frac{mp \mid i \overset{sl}{\longrightarrow} mp' \mid i'}{mp \mid i \overset{sl}{\sim\!\sim\!\sim\!\sim\!\sim\!>} mp' \mid i'}$$

$$\frac{mp_j \mid i \overset{pl_j}{\sim\!\sim\!\sim\!\sim\!\sim\!>} mp_j' \mid i_j' \qquad j = 1, 2}{mp_1 \mid mp_2 \mid i \overset{pl_1//pl_2}{\sim\!\sim\!\sim\!\sim\!\sim\!>} mp_1' \mid mp_2' \mid i'} \quad cond$$

Monitoring
$$\frac{mp \mid i \overset{pl}{\sim\!\sim\!\sim\!\sim\!\sim\!>} mp' \mid i'}{mp \mid mp_1 \mid i \overset{extl}{\longrightarrow} mp' \mid mp_1 \mid i'} \quad cond$$

Appropriate algebraic parameterized schemas are given for expressing the three steps; ultimately the specification of a system can be formally viewed as an adt specification which is an instantiation of a parameterized specification, say $SMoLCS(\ldots)$, where the parameters refer to various user defined aspects concerning dynamics and data. It is clearly possible then to reuse parts of specifications and also to structure hierarchically systems, corresponding to nested calling on different parameters, say $SMoLCS(\ldots, SMoLCS(\ldots), \ldots)$.

SMoLCS has been applied to significant case-studies, the most important being the specification of the underlying concurrent model in the formal definition of full

Ada (EEC-Map project The Draft Formal Definition of ANSI/STD 1815A see [5]). Among other applications, the description of a local area architecture in [4] and two-cases studies proposed by ENEL (Italian National Electricity Board) are concerning an hydro-electric central and a high-tension station for distribution of electric power.

For industrial applications it still lacks standardization and a complete toolset. A simulator, the SMoLCS rapid prototyping system based on RAP ([39]), has been in use since 1987, consisting of a tree-builder and a tree-walker. Currently a specification metalanguage for the SMoLCS specifications of concurrent systems and a related toolset is under development. The toolset includes a syntax-directed editor and a much more efficient redoing of the simulator equipped with a graphical interface for showing the results.

Following the SMoLCS approach it is possible to specify all CA architectures and the network CA_NET without implementing/simulating their concurrent features; such features may be abstractly specified by giving appropriate axioms for the various steps (the full specifications can be found in [8]). In this case it is important to note that the various assumptions on the data, on the buffer access and organization and on the parallelism are formalized by appropriate parameters of the parameterized adt *SMoLCS*; thus if we have the specification of the CA corresponding to a particular choice of such assumptions, say $CA(D_3,\ B_1,\ A_1,\ P_2)$, to get the specification for another choice, say $CA(D_3,\ B_3,\ A_1,\ P_2)$, it is sufficient to change the parameter corresponding to the buffer access (rules for the parallelism step).

3.5 Conditional Rewriting Logic

In [49] and, as application to object systems in [48], Meseguer advocates a formalism called "Conditional Rewriting Logic" as a unifying model of concurrency.

Meseguer work resembles much the SMoLCS approach (of which he was not aware when writing [49, 48]) but it differs in one fundamental point, as we will see. For example, the specifications of object systems in the second paper use for state configurations multisets of objects (processes) and information as in SMoLCS. Indeed it will be rather easy to understand that approach by comparing it to SMoLCS.

Seen from the specification viewpoint the proposed method can be viewed as a specialization of dynamic specifications in which:

- the transitions are seen as rewriting steps;
- a set of labelled conditional rewriting rules $(g\colon t \to t')$ is given, which constitute the proper axioms of the specification;
- a fixed set of conditional axioms is given defining the propagation of the rewriting steps by reflexivity, congruence, replacement and transitivity (in Meseguer's view a concurrent rewriting is characterized by the use of the replacement rule).

This work contains some very interesting ideas for theoretical foundations of concurrency, in particular the notion of semantics as a congruence over terms representing proofs; moreover it gives some very elegant insight into a categorical semantics. However for the moment those definitions are not able to embody significant observational semantics, which (personal communication) will constitute the subject of some future work.

The propagation axioms are the characterizing feature of Conditional Rewriting Logic; and indeed they justify the name of the approach (rewriting and logic). However, while being useful and elegant in some cases (for example in some application to Petri nets, which were the inspiring case), these axioms make this approach not convenient, in our view, for application to significant concurrent systems. The propagation rules imply that the actions corresponding to rewritings are not capabilities, but effective actions (silent moves in CCS). This implies that one cannot simulate labelled transitions (which are not explicitly supported) for representing capabilities. Hence what is a basic support for modular composition of open process modules (i.e., processes with capabilities toward the external world) is lacking. In most significant examples that we have encountered this makes specifications less modular than they should be.

Using the Conditional Rewriting Logic we can specify all CA variations and the network CA_NET; but the variations D_2 and D_3 are very hard to realize. The problem is that these specifications are very little modular due to the propagation axioms. We cannot first describe the rewritings of the processes and then those of the architectures, since if a process p can perform some rewriting only in some particular context, we have to specify only the architecture rewritings. So we cannot give a specification of the processes and then use it for all variations of CA; moreover we cannot use the specification of CA for specifying CA_NET. For the same reason the specification of CA choosing variations D_2 and D_3 are possible but very complicated, since we can only describe the rewritings of the architectures, otherwise the processes can perform some activity while they are communicated or stored in the buffer.

We should wait for more examples and applications, also remembering that probably the aims of that work are different from the classical specification of concurrent systems. Indeed the main application is currently related to the design, semantics and implementation of a specification language for concurrent modules. Quite interestingly, the semantics is driving the implementation down to the realization of the hardware architecture, which gives the project, in our view, a particular value as for the application of formal techniques to software engineering.

A somewhat related approach has been pursued in [28].

3.6 Stream Processing Functions

In various papers [22, 23, 24] and in projects M. Broy has developed since 1983 an approach to the formal specification of concurrent systems which is a combination of algebraic specifications, streams, predicate logic and functional programming.

The approach is denotational in nature: it provides a language and its semantics; thus it does not qualify as a typical algebraic formalism. However we briefly present it for matter of comparison; it may indeed use classic formalisms for the specification of abstract data types and it may apply first-order and temporal logic for describing properties of the agents.

The basic models are dataflow architectures and the structuring primitives are those typical for dataflows (which can be elegantly obtained as derived operators, because of the specification formalism and its semantics). Any other kind of concurrent architectures and of communication mechanisms have to be simulated.

There is a clear distinction between processes (agents) and data which may be defined as adt with some semantics; thus viewpoint **V1** is followed, since the concurrent architecture is not defined algebraically.

The overall specification is not algebraic and thus we cannot speak of adt specification. However the specifications which use first-order and temporal logic formulae identify classes of concurrent systems, thus VAS is supported.

Let us to see the approach in some more detail.

Broy's approach is built around a dataflow view of concurrent systems; consequently his basic semantic models are sets of continuous functions mapping tuples of streams to tuple of streams, see Sect. 1.1. Thus a process, called agent, has n input lines and m output lines. On every input line a finite or infinite sequence of data is transmitted to the agent and on every output line a finite sequence of data is generated by the agent. The input lines and output lines have internal (local) names that are used in a predicate for expressing the relationships between input and output.

A typical example is the following:

agent $store$ = **input stream data** d, **stream bool** b, **output stream data** r
first $b \Rightarrow r = store($**rest** d,**rest** $b)$
$\neg$**first** $b \Rightarrow r =$ **first** d & $store(d,$**rest**$b)$

Here d, b, r are used both as variables and as internal names.

Also nondeterministic agents are admitted, as in

agent $infinite$ = **output stream bool** r
$r =$ **true** & $r \vee r =$ **false** & r;

thus there are two possible output streams:

$r =$ **true** & **true** & ... and $r =$ **false** & **false** & ... .

From the examples we see that formulae are used for defining the streams of data, using of course primitive functions on streams like **first** and **rest**. Formulae may be first-order and also temporal logic formulas with operator like next, eventually and necessarily.

Sets of agents and recursive agents may be defined.

A semantics of a specification is given in the usual denotational way using basic domains and environments; the basic domains are data and agents:

$$D =_{def} DATA_{\perp} \cup STREAM(DATA)$$

$$AGENT =_{def} \{f \in [D^m \to D^n] \mid n, m \in \mathbb{N}\}$$

where $DATA$ is a set of atomic data objects, but it may also be given by an abstract data type specification; $DATA_\perp$ is the flat cpo associated with $DATA$. The semantics of a family of agent definitions is the set of all agent environments that fulfill the specification. For very technical reasons with any agent identifier a set of agents (functions) is associated instead of functions from tuples of streams into sets of tuples of streams.

An interesting feature of the formalism is that basic structuring operators like parallel composition "$\|$", sequential composition "$\cdot$" and feedback "C^i_j" may be obtained as derived operators. If a_1 and a_2 are defined by

agent a_1 = **input stream** x_1 **output stream** y_1 H_1 **end**
agent a_2 = **input stream** x_2 **output stream** y_2 H_2 **end**

then $a_3 = a_1 \| a_2$ is defined as

agent a_3 = **input stream** x_1, x_2 **output stream** y_1, y_2 $H_1 \wedge H_2$ **end**

Of course in this formalism algorithms may be described and thus so-called algorithmic agents can be defined.

Much importance is given to correctness, relative to safety (partial correctness) and liveness (robust correctness) properties, and to correctness of implementations, which are defined in a very elegant and simple way.

The CA architectures may be specified using this formalism, but the resulting specifications are not very natural since the CA architectures are based on communicating processes and so we have to realize their concurrent features using dataflows.

Broy's approach finds its most elegant applications in the specification of concurrent architectures which have essentially a dataflow structure. Indeed some nice examples of applications have been given, showing the potential applicability, at least for those architectures which are amenable to a dataflow structure. The method has also been applied in an EEC-MAP project (n. 785) in conjunction with industries, for giving a formal basis to the MASCOT method.

3.7 Algebraic Petri Nets

In the literature there are several papers presenting specification formalisms integrating Petri nets and algebraic specifications of adt's (in general not the elementary Petri nets introduced in Sect. 1.1 but e.g. predicate/transition or coloured nets); most of them follow the viewpoint **V1** but some one uses the algebraic techniques and results for handling, for example, nets composition or describing the firing rules. Here we briefly list some of the approaches known by the authors, but we do not claim that they are the only one. For a survey paper on this topic following viewpoint **V1** see e.g. [60]; where it is also shown that results about invariants could be obtained by classical algebraic results.

The Milan group has worked out a formalism "OBJSA Nets" combining Superposed Automata Nets (SA) with the possibility of defining the tokens and the transitions by means of parameterized algebraic OBJ specifications (see e.g. [15]).

OBJSA can be summarized as follows. The net structure is given as usual in superposed automata nets; the individuals flowing in the net consist of a name part, which models instances individuality and is not modified by transition firing, and a data part, which represents the data structure and can be modified by transition firing; the overall net system can be obtained through composition of the net models of its components (viewpoint **V1**).

Vautherin in [63] presents an algebraic version of coloured Petri nets, where the tokens of different colour are represented by elements of different sorts in the initial model of a specification of an adt and the structure of the net is given as usual (viewpoint **V1**), while Dimitrovici and Hummert in [33] show how to compose such nets by using categorical techniques.

In the following we report a simple example of these algebraic coloured Petri nets specifying a bounded buffer, containing natural numbers, organized as a queue. The tokens used in the net are defined by the initial model of a specification of the queues of natural numbers with the usual operations (Nil, InQueue, DeQueue, First and Length), which we do not report here. The schema of the net is graphically reported in Fig. 8; the arcs connecting places and transitions are labelled by open terms representing tokens; while the transitions are labelled by equations involving the variables appearing in such terms ($TKVar$); a transition may fire when for some evaluation V of the variables $TKVar$ satisfying its equation in the premise there are the tokens obtained by evaluating with V the relative terms.

Transition GET takes an element x from place $P1$ and puts within the queue q in place P, when the buffer is not full, i.e. when the equation $(Length(q) \leq n-1) = True$ holds. Transition $RETURN$ takes the first element out of the queue $First(q)$ and put it in the place $P2$, when the queue is not empty, i.e. when the equation $(Length(q) > 0) = True$ holds.

Fig. 8. An algebraic coloured net.

Bettaz in [19] and in [20] presents the so called "Algebraic Term Nets" and shows how such nets and their firing activity may be described by means of an algebraic specification of an adt (viewpoint **V2**).

Algebraic Petri nets has been also used as a basis for a specification metalanguage for distributed systems with real-time features [44].

4 Very Abstract Specifications of Concurrent Systems

In Sect. 2 we have already introduced the distinction between abstract and very abstract specifications of concurrent systems (shortly AS and VAS). An AS abstractly determines a concurrent system, i.e. it describes in an abstract way the system concurrent structure (which are its component processes and how they are arranged in the system) and activity; while a VAS abstractly determines a class of concurrent systems by giving only the relevant properties about their structure and activity.

Here we briefly list the algebraic approaches to specification of concurrency reported in Sect. 3 which can be extended to handle VAS's.

The Broy's approach, see Sect. 3.6, allows to express very abstract properties about the dynamic activity of classes of dataflow networks by using either first-order logics or various forms of temporal logics [22]; the last ones permit to formalize in a simple way liveness properties. However, this formalism does not allow to express requirements about the distributed structure of the networks. The dynamic requirements about a computer net given in Sect. 1.4 can be formalized using this approach, while the static ones cannot be considered. More importantly in this case we have also a notion of implementation between specifications of different abstraction levels; and in the literature there are examples of complete proofs of the correctness of some implementation (e.g. [24]).

Also in the framework of dynamic specifications, see Sect. 3.4, it is possible to give specifications of concurrent systems VAS both w.r.t. static and dynamic properties. Dynamic specifications are extended with the possibility of expressing very abstract properties about the dynamic activity of concurrent systems just by replacing conditional logic with more powerful ones. Initially first-order (infinitary) logic was considered, but it does not allow to express liveness properties; logics which integrates the combinators of temporal logic in the algebraic framework have been proposed in [29] and, more recently, in [58] whose "event logic" permits the formalization of the abstract properties of the activity of concurrent systems in terms of causal/temporal relationships among non-instantaneous events.

For what concerns the static (structure) properties, [57, 9] propose a subclass of the dynamic algebras, called "entity algebras", equipped with particular sorts, operations and predicates for describing the concurrent structure of the dynamic elements. Moreover, whichever logic for dynamic properties mentioned before (conditional, temporal, event, ...) may be extended with special predicates for formalizing abstract properties about the structure of dynamic elements ([57, 9]). Also for the dynamic VAS introduced above there is a notion of implementation extending that for specifications of static adt's of [66] (see [29, 57]); in these cases it is possible also to define particular kinds of implementations which e.g. preserve/refine the concurrent structure of a system, the atomicity grain of the activity of a system and so on. All the requirements about a computer net given in Sect. 1.4 can be simply formalized using dynamic VAS.

Acknowledgements. We thank H.Ehrig and F.De Cindio for various helpful comments.

References

1. A. Arnold and M. Nivat. Metric interpretations of infinite trees and semantics of non deterministic recursive programs. *TCS*, 11:181–205, 1980.
2. E. Astesiano, A. Giovini, and G. Reggio. Observational structures and their logic. *TCS*, 96, 1992.
3. E. Astesiano, G.F. Mascari, G. Reggio, and M. Wirsing. On the parameterized algebraic specification of concurrent systems. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Proc. TAPSOFT'85, Vol. 1*, number 185 in Lecture Notes in Computer Science, pages 342–358, Berlin, 1985. Springer Verlag.
4. E. Astesiano, F. Mazzanti, G. Reggio, and E. Zucca. Formal specification of a concurrent architecture in a real project. In *A Broad Perspective of Current Developments, Proc. ICS'85 (ACM International Computing Symposium)*, pages 185–195, Amsterdam, 1985. North-Holland.
5. E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini, P. Inverardi, E. Karlsen, F. Mazzanti, J. Storbank Pedersen, G. Reggio, and E. Zucca. The draft formal definition of Ada. Deliverable, CEC MAP project: The Draft Formal Definition of ANSI/STD 1815A Ada, 1986.
6. E. Astesiano and G. Reggio. An outline of the SMoLCS approach. In M. Venturini Zilli, editor, *Mathematical Models for the Semantics of Parallelism, Proc. Advanced School on Mathematical Models of Parallelism, Roma, 1986*, number 280 in Lecture Notes in Computer Science, pages 81–113, Berlin, 1987. Springer Verlag.
7. E. Astesiano and G. Reggio. SMoLCS-driven concurrent calculi. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proc. TAPSOFT'87, Vol. 1*, number 249 in Lecture Notes in Computer Science, pages 169–201, Berlin, 1987. Springer Verlag.
8. E. Astesiano and G. Reggio. A structural approach to the formal modelization and specification of concurrent systems. Technical Report 0, Formal Methods Group, Dipartimento di Matematica, Università di Genova, Italy, 1991.
9. E. Astesiano and G. Reggio. Entity institutions: Frameworks for dynamic systems. in preparation, 1992.
10. E. Astesiano, G. Reggio, and M. Wirsing. Relational specification and observational semantics. In *Proc. MFCS'86*, number 233 in Lecture Notes in Computer Science, pages 209–217, Berlin, 1986. Springer Verlag.
11. E. Astesiano and M. Wirising. An introduction to ASL. In L.G.L.T. Meertens, editor, *Program Specification and Transformation*, pages 343–365. North-Holland, 1987.
12. E. Astesiano and M. Wirising. Bisimulation in algebraic specifications. In M. Nivat and H. Ait-Kaci, editors, *Proc. of the Colloquium on Resolution of Equations in Algebraic Structures*, San Diego, 1989. Academic Press.
13. D. Austry and G. Boudol. Algebre de processus et synchronisation. *TCS*, 30:91–31, 1984.
14. J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, Cambridge, 1990.
15. E. Battiston, F. De Cindio, and G. Mauri. OBJSA nets: a class of high-level nets having objects as domains. In G. Rozemberg, editor, *Advances in Petri Nets*, number 340 in Lecture Notes in Computer Science, pages 20–43, Berlin, 1988. Springer Verlag.
16. L. Berardinello and F. De Cindio. A survey of basic net models and modular net classes. In G. Rozemberg, editor, *Advances in Petri Nets*, Lecture Notes in Computer Science, Berlin, 1992. Springer Verlag. To appear.

17. J.A. Bergstra, J. Heering, and P. Klint. ASF - an Algebraic Specification Formalism. Technical Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, 1987.
18. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information & Control*, 60(1/3):109–137, 1984.
19. M. Bettaz. An association of algebraic term nets and abstract data types for specifying real communication protocols. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification*, number 534 in Lecture Notes in Computer Science, pages 11–30, Berlin, 1991. Springer Verlag.
20. M. Bettaz. How to specify nondeterminism and true concurrency with algebraic term nets. Draft, 1992.
21. M. Bidoit. *PLUSS, un langage pour le developpment de specifications algebriques modulaires.* These d'Etat, Universite de Paris-Sud, 1989.
22. M. Broy. Specification and top down design of distributed systems. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Proc. TAPSOFT'85, Vol. 1*, number 185 in Lecture Notes in Computer Science, pages 4–28, Berlin, 1985. Springer Verlag.
23. M. Broy. Predicative specifications for functional programs describing communicating networks. *Information Processing Letters*, 25:2, 1987.
24. M. Broy. An example for the design of distributed systems in a formal setting: The lift problem. Techinical Report MIP P 8802, University of Passau, 1988.
25. M. Broy and M. Wirsing. Partial abstract types. *Acta Informatica*, 18:47–64, 1982.
26. R.M. Burstall and J.A. Goguen. Introducing institutions. In E. Clarke and D. Kozen, editors, *Logics of Programming Workshop*, number 164 in Lecture Notes in Computer Science, pages 221–255, Berlin, 1984. Springer Verlag.
27. V. Carchiolo, A. Faro, F. Minassale, and G. Scollo. Some topics in the design of the specification language LOTOS. In M. Paul and B. Robinet, editors, *Proc. 4th Int. Symp. on Programming*, number 167 in Lecture Notes in Computer Science, Berlin, 1984. Springer Verlag.
28. A. Corradini, G.L. Ferrari, and U. Montanari. Transition systems with algebraic structure as models of computation. In I. Guessarian, editor, *Proc. of 18-eme Ecole de Printemps en Informatique Theorique, Semantique du Parallelism*, number 469 in Lecture Notes in Computer Science, pages 185–222, Berlin, 1990. Springer Verlag.
29. G. Costa and G. Reggio. Abstract dynamic data types: a temporal logic approach. In A. Tarlecki, editor, *Proc. MFCS'91*, number 520 in Lecture Notes in Computer Science, pages 103–112, Berlin, 1991. Springer Verlag.
30. J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information & Control*, 54:70–120, 1982.
31. R. de Simone. Higher-level synchronising devices in Meije - SCCS. *TCS*, 37:245–267, 1985.
32. B.T. Denvir, W.T. Hardwood, M.J. Jackson, and M.J. Wray, editors. *The Analysis of Concurrent Systems.* Number 207 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1985.
33. 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*, number 534 in Lecture Notes in Computer Science, pages 52–73, Berlin, 1991. Springer Verlag.
34. H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An algebraic specification language with two levels of semantics. Technical Report 83-01, TUB, Berlin, 1983.
35. H. Ehrig, F. Parisi Presicce, P. Boehm, C. Rieckhoff, C. Dimitrovici, and M. Grosse-Rhode. Algebraic data type and process specifications based on projection spaces. In

D.Sannella and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, number 332 in Lecture Notes in Computer Science, pages 23–43, Berlin, 1988. Springer Verlag.
36. M. Grosse-Rhode and H. Ehrig. Transformation of combined data type and process specifications using projection algebras. In *Stepwise refinenment of distributed systems*, number 430 in Lecture Notes in Computer Science, pages 301–339, Berlin, 1990. Springer Verlag.
37. M. Hennessy. *Algebraic theory of processes.* The MIT Press, Cambridge, Massachusetts, 1988.
38. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, London, 1985.
39. H. Hussmann. Rapid prototyping for algebraic specifications: RAP system user's manual. Technical Report MIP P 8504, University of Passau, 1985.
40. I.S.O. LOTOS – A formal description technique based on the temporal ordering of observational behaviour. IS 8807, International Organization for Standardization, 1989.
41. G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Information Processing 77*, pages 471–475, Amsterdam, 1974. North-Holland.
42. S. Kaplan. Algebraic specification of concurrent systems. *TCS*, 9:90–115, 1989.
43. S. Kaplan and A. Pnueli. Specification and implementation of concurrently accessed data. In *Proc. STACS '87 (Symposium on Theoretical Aspects of Computer Science)*, number 247 in Lecture Notes in Computer Science, Berlin, 1987. Springer Verlag.
44. B. Kraemer. *Concepts, Syntax and Semantics of SEGRAS – A specification Language for Distributed Systems.* Oldenbourg verlag, Munchen, Wien, 1989.
45. K. Lodaya and P. S. Thiagarajan. A modal logic for a subclass of event structures. In T. Ottmann, editor, *Proceeding of ICALP'87*, number 267 in Lecture Notes in Computer Science, pages 290–303, Berlin, 1987. Springer Verlag.
46. S. Mauw and G.J. Veltink. An introduction to PSF_d. In J. Diaz and F. Orejas, editors, *Proc. TAPSOFT'89, Vol. 2*, number 352 in Lecture Notes in Computer Science, pages 272 – 285, Berlin, 1989. Springer Verlag.
47. S. Mauw and G.J. Veltink. A proof assistant for PSF. In K. Larsen and A. Skou, editors, *Proc. Third Workshop on Computer Aided Verification, Vol. 1*, pages 200 – 211, Aalborg, Denmark, 1991. The University of Aalborg.
48. J. Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa Canada, October 1990*, pages 101–115. ACM, 1990.
49. J. Meseguer. Rewriting as a unified model of concurrency. *TCS*, 96, 1992.
50. R. Milner. *A Calculus of Communicating Systems.* Number 92 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1980.
51. R. Milner. *Communication and concurrency.* Prentice Hall, London, 1989.
52. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes - Part I. *Information and Computation*, 1992. To appear.
53. D. Park. Concurrency and automata on infinite sequences. In *Proc. 5th GI Conference*, number 104 in Lecture Notes in Computer Science, Berlin, 1981. Springer Verlag.
54. G. Plotkin. A structural approach to operational semantics. Lecture notes, Aarhus University, 1981.
55. G. Plotkin. An operational semantics for CSP. In D. Bjorner, editor, *Proc. IFIP TC 2-Working conference: Formal description of programming concepts*, pages 199–223, Amsterdam, 1983. North-Holland.
56. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency*, number 224 in Lecture Notes in Computer Science, pages 510–584, Berlin, 1986. Springer Verlag.

57. G. Reggio. Entities: an istitution for dynamic systems. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification*, number 534 in Lecture Notes in Computer Science, pages 244–265, Berlin, 1991. Springer Verlag.
58. G. Reggio. Event logic for specifying abstract dynamic data types. In the same volume, 1992.
59. W. Reisig. *Petri nets: an introduction.* Number 4 in EATCS Monographs on Theoretical Computer Science. Springer Verlag, Berlin, 1985.
60. W. Reisig. Petri nets and algebraic specifications. *TCS*, 80:1–34, 1991.
61. B. Thomsen. A calculus of higher-order communicating systems. In *Proceeding of POPL Conference*, pages 143–154, 1989.
62. P. van Eijk. Tools for LOTOS, a Lotosfere overview. Memoranda Informatica 91-25, Universiteit Twente - Faculteit der Informatica, Enscede, 1991.
63. J. Vautherin. Parallel system specifications with coloured Petri nets and algebraic data types. In G. Rozemberg, editor, *Advances in Petri Nets*, number 266 in Lecture Notes in Computer Science, Berlin, 1987. Springer Verlag.
64. G. Winskel. Event structure semantics for CCS and related languages. In M. Nielsen and E.M. Schmidt, editors, *Proc. 9th ICALP*, number 140 in Lecture Notes in Computer Science, pages 561–576, Berlin, 1982. Springer Verlag.
65. G. Winskel. An introduction to event structures. In J.W. de Bakker, W.-P. de Roever, and G. Rozemberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 354 in Lecture Notes in Computer Science, pages 364–397, Berlin, 1989. Springer Verlag.
66. M. Wirsing. Algebraic specifications. In van Leeuwen Jan, editor, *Handbook of Theoret. Comput. Sci.*, volume B, pages 675–788. Elsevier, 1990.

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