Specification of Abstract Dynamic-Data Types: A Temporal Logic Approach [⋆]

Gerardo Costa and Gianna Reggio

Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Via Dodecaneso 35, 16146 Genova, Italy,
email: `costa, reggio @disi.unige.it`

Abstract

A concrete dynamic-data type is just a partial algebra with predicates such that for some of the sorts there is a special predicate defining a transition relation.

An abstract dynamic-data type (ad-dt) is an isomorphism class of such algebras. To obtain specifications for ad-dt's, we propose a logic which combines many-sorted 1st order logic with branching-time combinators.

We consider both an initial and a loose semantics for our specifications and give sufficient conditions for the existence of the initial models. Then we discuss structured specifications and implementation.

Keywords: abstract data types, algebraic specifications, specifications of dynamic systems, temporal logic, abstract dynamic-data types.

1 Introduction

Dynamic-data types are (modelled by) dynamic algebras, which are a particular kind of algebras with predicates. These, in turn, are just the algebraic structures that are needed to interpret many-sorted 1st order logic: a family of sets (the carriers) together with a set of operations and predicates on the carriers [30]; here the operations are partial in order to model situations like trying to get the first element of an empty list.

The distinguishing feature of dynamic algebras is that for some of the carriers (the dynamic ones) there are special ternary predicates $\longrightarrow$, where

[⋆] This work has been partially funded by EEC-HCM MEDICIS, ESPRIT-BR-WG COMPASS and by the project "Modelli della computazione e dei linguaggi di programmazione" (MURST - Italy).

$(d, l, d') \in \longrightarrow$, usually written as $d \xrightarrow{l} d'$, means that the element d may perform a transition labelled by l into the element d'. The label l is used to describe the interaction with the environment, by specifying both the conditions (on the environment) for the transition to become enabled and the transformations this transition induces on the environment. The elements d and d' are called dynamic elements, because we regard them as (descriptions of) entities that can evolve in time. Processes or concurrent/reactive systems are typical examples, but our framework can be applied to other situations as well; in [44], for instance, it is used to describe parts of a hydroelectric power station.

Following a well established pattern, see e.g. [40], we identify a dynamic entity with its (initial) state. Therefore *dynamic* sorts/carriers/elements could also be called *state* sorts/carriers/elements.

If we use a dynamic algebra to model processes, then we may have transitions corresponding to "send" and "receive" actions. A dynamic algebra for lists may have transitions corresponding to the tail operation; thus $\mathit{list} \xrightarrow{l} \mathit{Tail}(\mathit{list})$ is true, for some appropriate label l. Of course we are not forced to have such predicates: when modelling processes it is natural to use them (one could even say we need them); in the case of list we have a choice: we can use the (classical) static view, or a dynamic one (closer to the way we regard lists when programming within the imperative paradigm). The crucial point is that if we want to model, say, processes which can send and receive structured data, we can use a single algebra to describe both processes and data. Some of the examples in the main text should clarify this point.

The basic idea behind dynamic algebras is very simple. There are some technical problems; but they are orthogonal w.r.t. the dynamic features, as they concern partial operations, and have been dealt with in the literature, see [16,2] for instance. The name seems appropriate, even though it has already been used to denote structures for interpreting dynamic logic, see e.g. [42], which are different from the ones we consider here.

The question is whether dynamic algebras are of any use; we think the answer is yes. Indeed, they are a basic formal tool of the SMoLCS methodology, which has been used in practice, and for large projects, with success (see e.g. [7,5,10]). SMoLCS is a methodology for specifying dynamic systems that provides a framework for handling both ordinary (static) data types and dynamic data types. One of the main ideas in SMoLCS is indeed that this aim can be achieved within the algebraic framework developed by the adt-community, provided that transitions can be referred to (and this is precisely the role of transition predicates). There are other significant features in SMoLCS: great flexibility in defining the kind of composition and/or interaction of processes (there are no predefined operations); methodological guidelines for modular specifications of complex systems; software tools; ...; but they are not central to this paper; interested readers can refer to [7].

The logical language originally used in SMoLCS is (partial) many-sorted 1st order logic with equality and transition predicates. Such a language allows reasonable specifications for many properties of concurrent systems, however it becomes cumbersome and inadequate when dealing with properties involving the transitive closure of the transition relations such as (some) liveness or safety properties [38]. A really significant example would take up too much space here; a simple, but still interesting, one can be cooked up using buffers containing natural numbers.

To model buffers we use a dynamic algebra where the carriers are the set of natural numbers, a set of buffer states B, a set L of labels; the operations are *Empty*, *Put*, *Get*, *Remove*, O and I. *Empty* denotes the empty buffer, $Put(n, b)$ adds number n to buffer b; $Get(b)$ yields the "first" element in b, if any; $Remove(b)$ removes this first element from b returning a new buffer, $O(n)$ and $I(n)$ are the labels corresponding, respectively, to returning and receiving number n. Finally, $\longrightarrow$ consists of the triples: $b \xrightarrow{I(n)} Put(n, b)$, for all n and all b, and $b \xrightarrow{O(Get(b))} Remove(b)$, whenever b is not empty (see Ex. 3 for a more precise account).

As examples of properties we can consider:

(i) The buffers follow a LIFO policy, i.e.:
$Get(Put(n, b)) = n$ and $Remove(Put(n, b)) = b$.
(ii) If b is non-empty, then there is an elementary transition from b to $Remove(b)$ corresponding to "output $Get(b)$". Using the transition predicate, we can phrase this by saying that $b \xrightarrow{O(Get(b))} Remove(b)$ is true.
(iii) The buffers will return any number that they receive.
(iv) The buffers have the capability of returning any number, say n, that they receive and maintain this capability until n is actually delivered.

Condition (i) is standard and does not need any comment. (iii) is a liveness constraint: once a buffer b inputs n it will evolve (through input/output transitions) in such a way that at a certain "state" (or moment) n will be output. Notice one important difference between (i) and (iii): the first specifies the structure of our buffers, while the other specifies their behaviour, without constraining the internal structure. (iv) is a weaker form of (iii): once a buffer b inputs n it will evolve in such a way that at any moment either n is output or another state can be reached in which n can be output. Notice also that (iv) is about the future capabilities of the buffers to perform some output actions, so it does not require anything about possible external receivers of the output number; while (iii) implicitly requires the existence of a receiver.

Finally, one way of reading (ii) is: if b is nonempty then it can always output the last (stored) value; thus we have an example of a simple safety property.

Properties (i) and (ii) can be easily expressed using a 1st order logic (with transition predicates). This is not the case for the other two properties. Let us consider (iv): the natural way of expressing it refers, more or less explicitly, to the (future) behaviour of buffers, behaviour that is usually described by transition sequences. In other words, the straightforward formalization of (iv) using the "usual language of mathematics" would look like this (if we assume for simplicity that a buffer always holds distinct elements):

$$\begin{array}{ll}(\mathrm{iv}') \text{ if } b_0 \xrightarrow{I(n)} b_1 \text{ then } & \forall\, k \geq 2,\ \forall\, b_2, \ldots, b_k \in B, \forall\, l_2, \ldots, l_{k-1} \in L \text{ s.t.} \\ & \quad b_1 \xrightarrow{l_1} b_2,\, b_2 \xrightarrow{l_2} b_3,\, \ldots,\, b_{k-1} \xrightarrow{l_{k-1}} b_k, \\ & \text{either } \exists\, j \;\; 1 \leq j \leq k-1 \text{ s.t. } l_i = O(n) \text{ or} \\ & \text{there exist } b'_1, \ldots, b'_m \in B,\, l'_1, \ldots, l'_{m-1} \in L \text{ s.t.} \\ & \quad b_k \xrightarrow{l'_1} b'_1,\, \ldots,\, b'_{m-1} \xrightarrow{O(n)} b'_m.\end{array}$$

It is hard to derive a simple 1st order formula from (iv′)! This is one of the reasons why we decided to use a "richer" logic: one well suited to express properties such as (iii) and (iv), but also (i) and (ii).

Various modal and temporal logics have been proposed as a tool for specifying properties of concurrent/reactive systems. As pointed out in [49] modal logics allow to express in concise form properties which refer to *single* transitions. In most cases, multi-modal logics (where modalities are labelled by actions) are used, interpreting them over labelled transition systems (which generalize Kripke frames). However, when the main interest is in describing the on-going behaviour of a system (as in our case), temporal logics are preferred as they are interpreted over paths (in transition systems) and such paths indeed describe the evolution of the process/system in question.

Linear temporal logics refer to *single* paths (thus a formula is satisfied by a set of paths iff it is satisfied by each path in the set), while *branching-time* temporal logics refer to *sets of paths* arranged in a tree (or part of a tree) thus taking the branching structure of the behaviour into account. (For an overview of modal and temporal logics see for instance [49] and [25]).

Manna and Pnueli have, over the years, carried out the most extensive investigation in using temporal logics for describing reactive systems. Their approach (see [37,38]) has been followed, with minor changes, by many authors. Essentially, they model the behaviour of systems by maximal (i.e. non-extendable) *sequences of states*. The j-th state in the sequence describes the overall state of the system after j "steps" of activity. The logic they use (in its more recent formulation, see [38]) is a 1st order linear temporal logic; with minor differences it is the same logic presented in [33,25,1]. In this logic, state-formulae are the basic building blocks; they are (or can be regarded as) ordinary 1st order formulae and describe the properties of the system under investigation at a given instant in time. Temporal formulae are obtained from state formulae by using temporal combinators (such as "henceforth", "eventually", "at the next instant", ...), together with classical propositional connectives and quantifiers; they allow to express, for instance, safety and liveness properties. Semantically, states (and the interpretation of non-logical symbols) provide the equivalent of classical 1st order structures, while the interpretation of the temporal combinators refers to a maximal sequence of states (recall that it is a *linear* temporal logic). At the semantic level, therefore, the basic picture is a *sequence of 1st order structures*. As there is *one* system to describe/specify these structures must have a lot in common; on the other hand something *must* change, as times flows. (This is also connected with some problems about the meaning of 1st order quantification within temporal logic, see [29].) A solution is obtained by considering just a single 1st order structure, but allowing *some* function and/or predicate symbols to be *flexible* (or local): their interpretation (or the assignment of values to them, in the case of variables) is time-dependent (i.e. state-dependent), moreover quantification over flexible variables is distinguished from quantification on ordinary (*rigid*/global) variables. Nevertheless, in this approach 1st order and temporal features do not mix really well; in particular the temporal dimension remains *external* to the 1st order world.

The 1st order logic used in specifications of concurrent/reactive systems following the SMoLCS methodology suggested a different approach, in which individual elements, operations, predicates and *transitions* (modelled by a special predicate) are all contained within a single algebraic structure: a dynamic algebra. (As a side benefit, the semantic problems related to quantification vanish.) In other words, the evolution of a system is not described by a sequence of snapshots of a structure with "flexible components", but by a sequence of elements belonging to a single algebra and related by transition predicates. We shall be more precise on this point in Sect. 4.

We use a branching-time logic, instead of the simpler linear one, because it allows to express, in a natural way, properties about the choices available at a given moment of time.

Another important difference between our approach and the one by Manna and Pnueli (and other authors) is that rather than specifying single systems we specify *types* of systems (dynamic-data types), usually having several components of other (possible dynamic) types; each type is characterized by its properties expressed through logical axioms. For instance, rather than specifying a computer network, we specify a type "computer network", together with the types corresponding to the components of a network: a type "node", a type “server”, a type “storage unit”, ...

Many authors have limited the strength of the logic they use in order to preserve nice properties such as decidability or, at least, existence of complete deductive systems. We have privileged flexibility and expressiveness instead; therefore we have a full 1^{st} order branching-time logic. The price we pay for this is incompleteness: in our logic validity is not even semi-decidable.

Actually, the logic presented here does not have all the desirable features: past-time operators are not included, for simplicity. It has been argued, see e.g. [37], that they allow to write specifications which are simpler and more natural than those using future-time operators only. We share this view and in the applications we actually use an extension of the logic presented here, containing the usual operators referring to the past: since, last-time, sometime, ..., see e.g. [27].

As stated above, our choices have been motivated by previous experience of the problems involved in the specifications of concurrent systems [6,7,11]. In this paper we show that the logical framework we propose is sound. Not only it corresponds to an institution [17] but, more importantly, we have been able to extend to our setting, and in a natural way, the main concepts and results concerning “classical” specifications of abstract data types (see e.g.: [53]). In other words, we may go through the well known basic concepts and results about abstract data types replacing algebras with dynamic algebras and 1^{st} order logic with our logic. In Sect. 5 we exemplify this procedure on structured specifications and implementation of specifications.

The formalism presented in this paper is based on the institution of partial algebras with predicates, but this choice is not essential. Indeed in [18] it is shown how to define an operation that given an “appropriate” algebraic institution (total, order-sorted, non-strict, ...) produces the corresponding institution of dynamic specifications.

In Sect. 2 we summarize the main definitions and facts about partial algebras with predicates and in Sect. 3 we introduce dynamic algebras. In Sect. 4 we define our logical language and introduce dynamic specifications; moreover we give some results concerning initial models. Sect. 5 deals with structured specifications and implementations of specifications. Finally, Sect. 6 contains some concluding remarks and comparisons with other approaches.

This paper supersedes [19], whose material is contained in Sect. 3 and 4.

2 Partial Algebras with Predicates

Here we summarize the main definitions and facts about *partial algebras with predicates*, which are derived from the partial algebras of Broy and Wirsing ([16]) and from the algebras with predicates of Goguen and Meseguer ([30]).

A *predicate signature* (shortly, a *signature*) is a triple $\Sigma = (SRT, OP, PR)$, where

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

We shall write $Op\colon srt_1 \times \ldots \times srt_n \to srt$ for $Op \in OP_{srt_1 \ldots\, srt_n, srt}$ and $Pr\colon srt_1 \times \ldots \times srt_n$ for $Pr \in PR_{srt_1 \ldots\, srt_n}$; but also $Op \in OP$ and $Pr \in PR$ (when sorts are irrelevant).

A *partial Σ-algebra with predicates* (shortly a *Σ-algebra*) is a triple

$$A = (\{A_{srt}\}_{srt \in SRT}, \{Op^A\}_{Op \in OP}, \{Pr^A\}_{Pr \in PR})$$

consisting of the *carriers*, the *interpretation of the operation symbols* and the *interpretation of the predicate symbols*. More precisely:

- if $srt \in SRT$, then A_{srt} is a set;
- if $Op\colon srt_1 \times \ldots \times srt_n \to srt$, then $Op^A\colon A_{srt_1} \times \ldots \times A_{srt_n} \to A_{srt}$ is a partial function;
- if $Pr\colon srt_1 \times \ldots \times srt_n$, then $Pr^A \subseteq A_{srt_1} \times \ldots \times A_{srt_n}$.

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

The class of all the Σ-algebras is denoted by $\mathbf{PPAlg}_\Sigma$.

Assume that $\mathcal{X}$ is a given infinite denumerable set of variable symbols and let Σ be a signature with set of sorts SRT. A *sort assignment* for $\mathcal{X}$ w.r.t. Σ is a partial function $X\colon \mathcal{X} \to SRT$; in what follows we shall also regard X as an SRT-indexed family $\{X_{srt}\}_{srt \in SRT}$, where

$$X_{srt} = \{\chi \mid \chi \in \mathcal{X} \text{ and } X(\chi) = srt\}.$$

Given a sort assignment X, the *term algebra* $T_\Sigma(X)$ is the Σ-algebra defined as follows, using T to denote $T_\Sigma(X)$:

- $x \in X_{srt}$ implies $x \in T_{srt}$;

– $Op \in OP_{\Lambda,srt}$ implies $Op \in T_{srt}$;
– $t_i \in T_{srt_i}$ for $i = 1, \dots, n$ and $Op \in OP_{srt_1 \dots srt_n, srt}$ imply
 $Op(t_1, \dots, t_n) \in T_{srt}$;
– $Op^T(t_1, \dots, t_n) = Op(t_1, \dots, t_n)$ for all $Op \in OP$;
– $Pr^T = \emptyset$ for all $Pr \in PR$.

If $X_{srt} = \emptyset$ for all $srt \in SRT$, then $T_\Sigma(X)$ is simply written T_Σ and its elements are called *ground terms*.

If $A \in \mathbf{PPAlg}_\Sigma$, a *variable evaluation* $V: X \to A$ is a sort-respecting assignment of values in A to *all* the variables in X. If $t \in T_\Sigma(X)$, the *interpretation of* t *in* A *w.r.t.* V is denoted by $t^{A,V}$ and given as usual; note however that here it may be undefined. When t is a ground term, we use the notation t^A.

A Σ-algebra A is *term-generated* iff for all $srt \in SRT$ and all $a \in A_{srt}$ there exists $t \in (T_\Sigma)_{srt}$ such that $a = t^A$.

In what follows we assume that sorts and arities are respected and also that our algebras have *nonempty carriers* (as this applies to term algebras as well, we have an implicit assumption on signatures: that they contain "enough constants symbols").

If A and B are Σ-algebras, a *homomorphism* h from A into B, written $h: A \to B$, is a family of *total* functions $h = \{h_{srt}: A_{srt} \to B_{srt}\}_{srt \in SRT}$ s.t.:

– for all $Op \in OP$:
 if $Op^A(a_1, \dots, a_n)$ is defined, then so is $Op^B(h_{srt_1}(a_1), \dots, h_{srt_n}(a_n))$
 and moreover $h_{srt}(Op^A(a_1, \dots, a_n)) = Op^B(h_{srt_1}(a_1), \dots, h_{srt_n}(a_n))$;
– for all $Pr \in PR$: if $Pr^A(a_1, \dots, a_n)$, then $Pr^B(h_{srt_1}(a_1), \dots, h_{srt_n}(a_n))$.

An *isomorphism* is a homomorphism that admits an inverse.

It is well known that there are several possible definitions of homomorphism between partial algebras. The one chosen here guarantees the properties formalized in Prop. 5 and 6 (see Sect. 3) and that our specification framework is an institution (see Sect. 4.4).

Algebras and homomorphisms form a category, still denoted by $\mathbf{PPAlg}_\Sigma$: the identity homomorphism is the family of identity functions and the composition is the composition component by component.

The interpretation of a formula of (many-sorted) 1st order logic with equality (with operation and predicate symbols belonging to Σ) in a Σ-algebra A is given as usual, but:

 for t_1, t_2 of the same sort, $t_1 = t_2$ is *true in* A *w.r.t. a variable evaluation*

V iff $t_1^{A,V}$ and $t_2^{A,V}$ are both defined and equal in A (we say that $=$ denotes "existential equality").

We write $A, V \models \theta$ when the interpretation of the formula θ in A w.r.t. V yields true; moreover, θ is *valid* in A (written $A \models \theta$) whenever $A, V \models \theta$ for all evaluations V. Usually we simply write $D(t)$ for $t = t$ and use it to require that the interpretation of t is defined.

Given a class of Σ-algebras $\mathcal{C}$, an algebra I is *initial* in $\mathcal{C}$ iff $I \in \mathcal{C}$ and for all $A \in \mathcal{C}$ there is a unique homomorphism $h^A\colon I \to A$; notice that the initial algebra is unique up to isomorphisms.

Proposition 1 *If I is initial in $\mathcal{C}$, then for all ground terms $t_1, \ldots, t_n$ and all predicates $Pr \in PR$:*

- *$I \models t_1 = t_2$ iff for all $A \in \mathcal{C}$: $A \models t_1 = t_2$; thus*
  $I \models D(t_1)$ iff for all $A \in \mathcal{C}$: $A \models D(t_1)$;
- *$I \models Pr(t_1, \ldots, t_n)$ iff for all $A \in \mathcal{C}$: $A \models Pr(t_1, \ldots, t_n)$.* $\square$

The condition above on $D(t)$ implies that, in general, the term algebra T_Σ is not initial in the class $\mathbf{PPAlg}_\Sigma$.

Finally, if $\Sigma \subseteq \Sigma'$ and A is a Σ'-algebra, then the *restriction of A to Σ* is the Σ-algebra denoted by $A_{|\Sigma}$ and given by:

- $(A_{|\Sigma})_{srt} = A_{srt}$ for all sorts srt of Σ,
- $Op^{A_{|\Sigma}} = Op^A$ for all operation symbols Op of Σ,
- $Pr^{A_{|\Sigma}} = Pr^A$ for all predicate symbols Pr of Σ.

3 Dynamic Algebras

Definition 2

- *A* dynamic signature *$D\Sigma$ is a pair (Σ, DS) where:*
  - *$\Sigma = (SRT, OP, PR)$ is a predicate signature,*
  - *$DS \subseteq SRT$ (the elements in DS are the* dynamic sorts*, i.e. the sorts of dynamic elements),*
  - *for all $ds \in DS$ there exist a sort $lab(ds) \in SRT - DS$ and a predicate symbol $\_ \xrightarrow{\_} \_\colon ds \times lab(ds) \times ds \in PR$.*
- *A* dynamic algebra *on $D\Sigma$ (shortly a $D\Sigma$-algebra) is just a Σ-algebra; the term algebra $T_{D\Sigma}(X)$ is just $T_\Sigma(X)$.*

Note 1 *In this paper, for some of the operation and predicate symbols, we use a mixfix notation. This is explicit in the definition of the signatures; for* *instance,* $\_ \xrightarrow{\_} \_ : ds \times lab(ds) \times ds \in PR$ *means that we shall write* $t \xrightarrow{t'} t''$ *instead of* $\longrightarrow (t, t', t'')$*; i.e. terms of appropriate sorts replace underscores.*

If DA is a $D\Sigma$-algebra and $ds \in DS$, then: the elements of sort ds, the elements of sort $lab(ds)$ and the interpretation of the predicate $\_ \xrightarrow{\_} \_$ correspond, respectively, to the states, the labels and the transitions of a labelled transition system. The different possible evolutions of the dynamic elements are represented by the *maximal labelled paths*, i.e. maximal sequences of states and labels of the form

$$d_0 \xrightarrow{l_0}{}^{DA} d_1 \xrightarrow{l_1}{}^{DA} d_2 \ldots .$$

We denote by $PATH(DA, ds)$ the set of such paths for the dynamic elements of sort ds. More precisely, $PATH(DA, ds)$ is the set of all sequences having either of the two forms below:

(1) $d_0\ l_0\ d_1\ l_1\ d_2\ l_2 \ldots d_n\ l_n \ldots$ (infinite path)
(2) $d_0\ l_0\ d_1\ l_1\ d_2\ l_2 \ldots d_n\ l_n \ldots d_k \quad k \geq 0$ (finite path)
where for all $n \in \mathbb{N}$: $d_n \in DA_{ds}$, $l_n \in DA_{lab(ds)}$ and $(d_n, l_n, d_{n+1}) \in \rightarrow^{DA}$; moreover, in (2) for no d, l: $(d_k, l, d) \in \rightarrow^{DA}$ (there are no transitions starting from the final state of a finite path).

If σ is either (1) or (2) above, then

- $S(\sigma)$ denotes the first element of σ: d_0;
- $L(\sigma)$ denotes the second element of σ: l_0 (if it exists);
- $\sigma_{|n}$ denotes the path $d_n\ l_n\ d_{n+1}\ l_{n+1}\ d_{n+2}\ l_{n+2} \ldots$ (if it exists).

In what follows $D\Sigma$ will denote a generic dynamic signature (Σ, DS), where $\Sigma = (SRT, OP, PR)$; moreover we often omit the canonical sorts and predicates and write

sorts SRT'
dsorts DS
opns
    OP
preds
    PR'

for the dynamic signature (Σ, DS), where Σ is:
$(SRT' \cup DS \cup \{lab(ds) \mid ds \in DS\}, OP,$
$PR' \cup \{\_ \xrightarrow{\_} \_ : ds \times lab(ds) \times ds \mid ds \in DS\})$.

Example 3 Buffers containing natural values organized in a LIFO way
Consider the following dynamic signature:

sig BUFΣ =

sorts *nat*
dsorts *buf*
opns

$0\colon \to nat$
$Succ\colon nat \to nat$
$Empty\colon \to buf$
$Put\colon nat \times buf \to buf$
$Get\colon buf \to nat$
$Remove\colon buf \to buf$
$I, O\colon nat \to lab(buf)$

and the (term-generated) algebra BUF given as follows.

- $BUF_{nat} = \mathbb{N}$.
- BUF_{buf} *and the interpretation of the operations Empty, Put, Get and Remove are respectively the set of stacks of natural numbers and the usual operations empty stack, adding an element, getting the first element and removing the first element. These stacks are precisely our buffers.*
- *The elements built by the two operations I and O label the transitions corresponding to the actions of receiving and returning a value, respectively.*
- *If we assume that the buffers are bounded and can contain k elements at most, then the interpretation of* $\longrightarrow$ *in BUF is the relation consisting of the following triples (here and below the interpretation of a [predicate/ operation] symbol Symb in BUF,* $Symb^{BUF}$*, is simply denoted by Symb):*
  $b \xrightarrow{I(n)} Put(n, b)$ *for all n and all b having* $k-1$ *elements at most,*
  $b \xrightarrow{O(Get(b))} Remove(b)$ *for all* $b \neq Empty$.
  If we assume, instead, that the buffers are unbounded, then $\longrightarrow$ *consists of the triples:*
  $b \xrightarrow{I(n)} Put(n, b)$ *for all n and all b,*
  $b \xrightarrow{O(Get(b))} Remove(b)$ *for all* $b \neq Empty$.

The complete behaviour of a bounded buffer, with $k = 2$*, which is initially empty (this buffer is represented by the term Empty) is given by the tree represented in Fig. 1. Notice that the paths, starting from the root, in this three are exactly the elements in* $PATH(BUF, buf)$ *whose initial element is Empty. Each path describes a possible behaviour of the initially empty buffer; the transitions labelled by* $I(\ldots)$ *and* $O(\ldots)$ *are, of course, triggered by requests from the outside world. In Sect. 5 we shall present examples (of specifications) where processes interact through buffers similar to the ones considered here.*

Definition 4 *Let DA and DA′ be DΣ-algebras; a* (dynamic) homomorphism $h\colon DA \to DA'$ *is just a homomorphism of partial algebras with predicates, i.e. a homomorphism from DA into DA′ as Σ-algebras.*

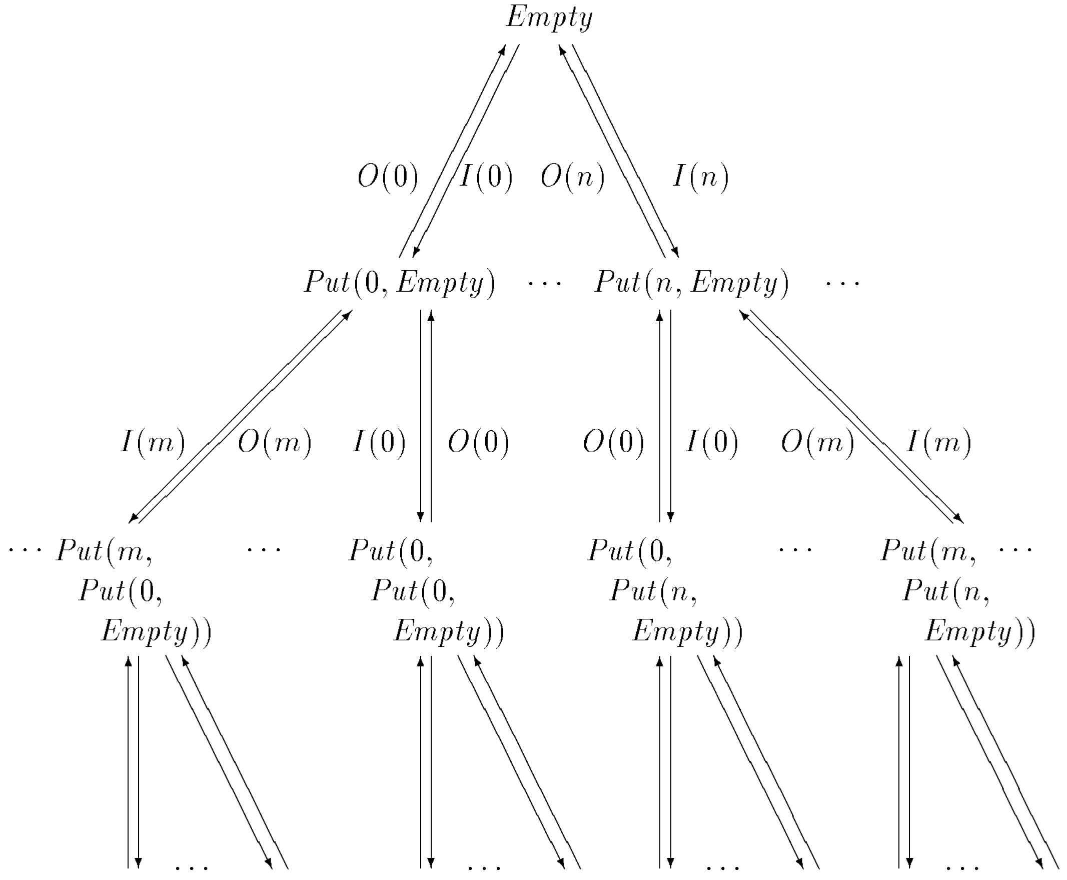

Fig. 1. The execution tree (in condensed format) associated with the empty unbounded buffer.

It is easy to see that, for each signature $D\Sigma$, the class of all $D\Sigma$-algebras and the dynamic homomorphisms form a category, that we denote by $\mathbf{DAlg}_{D\Sigma}$.

Proposition 5 *Let $h\colon DA \to DA'$ be a homomorphism; for all d, l, $d' \in DA$: if $d \xrightarrow{l}{}^{DA} d'$ then $h(d) \xrightarrow{h(l)}{}^{DA'} h(d')$.* $\square$

Informally: homomorphisms preserve the activity of the dynamic elements. The proof is obvious from the definition of homomorphism.

Proposition 6 *If $\mathcal{D}$ is a class of dynamic $D\Sigma$-algebras and DI is initial in $\mathcal{D}$, then $DI \models t \xrightarrow{t'} t''$ iff for all $DA \in \mathcal{D}$: $DA \models t \xrightarrow{t'} t''$.* $\square$

Informally: DI is an algebra of $\mathcal{D}$ where each dynamic element has the minimum amount of activity. The proof follows from the properties of the initial elements in the category of partial algebras with predicates (see Prop. 1).

4 Specifications of Abstract Dynamic-Data Types

Following a widely accepted idea (see e.g. [53]) a (static) *abstract data type* (shortly adt) is an isomorphism class of Σ-algebras and it is usually given by a *simple specification*, i.e. a pair $sp = (\Sigma, AX)$, where Σ is a signature and AX a set of 1st order formulae on Σ (the *axioms* of sp) representing the properties of the adt. The *models* of sp are precisely the Σ-algebras which satisfy the axioms in AX; more precisely, the class of models of sp is:

$$Mod(sp) = \{A \mid A \text{ is a } \Sigma\text{-algebra and for all } \theta \in AX \colon A \models \theta\}.$$

In the *initial algebra approach* sp defines the adt consisting of the (isomorphism class of the) initial elements of the class $Mod(sp)$. The principal motivation for this choice is that initial models enjoy the properties mentioned in Prop. 1. Not all specifications admit initial models; their existence is guaranteed, however, if the formulae in AX are *positive conditional axioms*, i.e. they have the form $\wedge_{i=1,\ldots,n}\ \alpha_i \supset \alpha_0$, where, for all i, α_i is an *atom*, i.e. it is either $Pr(t_1, \ldots, t_n)$ or $t = t'$.

In the *loose* approach, instead, sp is viewed as a description of the main properties of an adt; thus it represents a class, consisting of all the adt's satisfying the properties expressed by the axioms (more formally: the class of all isomorphism classes included in $Mod(sp)$).

The above definition of adt can be easily adapted to the dynamic case: an *abstract dynamic-data type* (shortly *ad-dt*) is an isomorphism class of $D\Sigma$-algebras. In order to extend the definition of specification, the problem is choosing the appropriate logical framework. We have already discussed some of the problems in the introduction, therefore we first define our logic and then comment on it.

4.1 The Logic

Recall that $D\Sigma = (\Sigma, DS)$ and $\Sigma = (SRT, OP, PR)$; moreover let X be a fixed sort assignment as in Sect. 2.

Definition 7 *The set $F_{D\Sigma}(X)$ of* dynamic formulae *and the auxiliary sets $P_{D\Sigma}(X, ds)$ of* path formulae *of sort $ds \in DS$ (on $D\Sigma$ and X) are inductively defined as follows (where t_1, ..., t_n denote terms of appropriate sort and we assume that sorts are respected):*

dynamic formulae

– $Pr(t_1, \ldots, t_n) \in F_{D\Sigma}(X)$ *if* $Pr \in PR$

- $t_1 = t_2 \in F_{D\Sigma}(X)$
- $\neg\, \phi_1,\, \phi_1 \supset \phi_2 \in F_{D\Sigma}(X)$ *if* $\phi_1, \phi_2 \in F_{D\Sigma}(X)$
- $\forall\, x \,.\, \phi \in F_{D\Sigma}(X)$ *if* $\phi \in F_{D\Sigma}(X)$, $x \in X$
- $\triangle(t, \pi) \in F_{D\Sigma}(X)$ *if* $t \in T_{D\Sigma}(X)_{ds}$, $\pi \in P_{D\Sigma}(X, ds)$, $ds \in DS$

path formulae

- $[\, \lambda x \,.\, \phi\,] \in P_{D\Sigma}(X, ds)$ *if* $x \in X_{ds}$, $\phi \in F_{D\Sigma}(X)$
- $\langle \lambda x \,.\, \phi \rangle \in P_{D\Sigma}(X, ds)$ *if* $x \in X_{lab(ds)}$, $\phi \in F_{D\Sigma}(X)$
- $\neg\, \pi_1,\, \pi_1 \supset \pi_2 \in P_{D\Sigma}(X, ds)$ *if* $\pi_1, \pi_2 \in P_{D\Sigma}(X, ds)$
- $\forall\, x \,.\, \pi \in P_{D\Sigma}(X, ds)$ *if* $\pi \in P_{D\Sigma}(X, ds)$, $x \in X$
- $\pi_1 \,\mathcal{U}\, \pi_2 \in P_{D\Sigma}(X, ds)$ *if* $\pi_1, \pi_2 \in P_{D\Sigma}(X, ds)$.

Remark 8 *The symbols* $[\quad]$ *and* $\langle\ \rangle$ *that appear in path formulae are just brackets and do not represent modalities.*

Definition 9 *Let DA be a $D\Sigma$-dynamic algebra and $V\colon X \to DA$ be a variable evaluation (i.e. an SRT-family of total functions). We now define by multiple induction when a formula $\phi \in F_{D\Sigma}(X)$* holds in DA under V *(written $DA, V \models \phi$) and when a formula $\pi \in P_{D\Sigma}(X, ds)$* holds in DA on a path $\sigma \in PATH(DA, ds)$ under V *(written $DA, \sigma, V \models \phi$). Recall that the interpretation of a term t in DA w.r.t. V is denoted by $t^{DA,V}$ and that $S(\sigma)$, $L(\sigma)$ and $\sigma_{|i}$ have been defined in Sect. 3.*

dynamic formulae

- $DA, V \models Pr(t_1, \ldots, t_n)$ *iff* $(t_1^{DA,V}, \ldots, t_n^{DA,V}) \in Pr^{DA}$
- $DA, V \models t_1 = t_2$ *iff* $t_1^{DA,V} = t_2^{DA,V}$
  (both sides must be defined and equal)
- $DA, V \models \neg\, \phi$ *iff* $DA, V \not\models \phi$
- $DA, V \models \phi_1 \supset \phi_2$ *iff* *either* $DA, V \not\models \phi_1$ *or* $DA, V \models \phi_2$
- $DA, V \models \forall\, x \,.\, \phi$ *iff* *for all* $v \in DA_{srt}$, *with* srt *sort of* x,
  $DA, V[v/x] \models \phi$
- $DA, V \models \triangle(t, \pi)$ *iff* $t^{DA,V}$ *is defined and*
  for all $\sigma \in PATH(DA, ds)$, *with* ds *sort of* t,
  if $S(\sigma) = t^{DA,V}$ *then* $DA, \sigma, V \models \pi$

path formulae

- $DA, \sigma, V \models [\, \lambda x \,.\, \phi\,]$ *iff* $DA, V[S(\sigma)/x] \models \phi$
- $DA, \sigma, V \models \langle \lambda x \,.\, \phi \rangle$ *iff* *either* $DA, V[L(\sigma)/x] \models \phi$ *or* $L(\sigma)$ *is not defined*
- $DA, \sigma, V \models \neg\, \pi$ *iff* $DA, \sigma, V \not\models \pi$
- $DA, \sigma, V \models \pi_1 \supset \pi_2$ *iff* *either* $DA, \sigma, V \not\models \pi_1$ *or* $DA, \sigma, V \models \pi_2$
- $DA, \sigma, V \models \forall\, x \,.\, \pi$ *iff* *for all* $v \in DA_{srt}$, *with* srt *sort of* x,
  $DA, \sigma, V[v/x] \models \pi$
- $DA, \sigma, V \models \pi_1 \,\mathcal{U}\, \pi_2$ *iff* *there exists* $j > 0$ *s.t.* $\sigma_{|j}$ *is defined,*

$DA, \sigma_{|j}, V \models \pi_2$ *and*
for all i *s.t.* $0 < i < j$ $DA, \sigma_{|i}, V \models \pi_1$.

A formula $\phi \in F_{D\Sigma}(X)$ *is* valid *in* DA *(written* $DA \models \phi$*) iff* $DA, V \models \phi$ *for all evaluations* V*. If* Γ *is a set of formulae and every* ϕ *in* Γ *is valid in* DA*, then* DA *is a* model *for* Γ*.*

Remark 10 *Dynamic formulae include the usual (static) formulae of many-sorted 1st order logic with equality; if* $D\Sigma$ *contains dynamic sorts, they include also formulae built with the transition predicates.*

Notice that the formulae of our language are the dynamic ones; indeed the axioms of our specifications are dynamic formulae. Path formulae are just an ingredient, though an important one.

The formula $\triangle(t, \pi)$ *can be read as "for every path* σ *starting from the state denoted by* t*, (the path formula)* π *holds on* σ*". We have borrowed* $\triangle$*, and* $\bigtriangledown$ *below, from [48]. We anchor these formulae to states, following the ideas in [37]. The difference is that we do not model a single system but, in general, a group of systems, so there is not a single initial state but several of them, hence the need for an explicit reference to states (through terms) in the formulae built with* $\triangle$ *and* $\bigtriangledown$*.*

The formula $[\lambda x \,.\, \phi]$ *holds on a path* σ *whenever* ϕ *holds at the first state of* σ*; similarly the formula* $\langle \lambda x \,.\, \phi \rangle$ *holds on* σ *if either* σ *consists of a single state or* ϕ *is true of the first label of* σ*. The need for both state and edge formulae has been already discussed in [35].*

Here labels can be arbitrarily complex, one can also have operations to compose/decompose them. This allows to model in a direct way the interaction of an open system with its environment, or between parts of a system.

Finally, $\mathcal{U}$ *is the so called strong future until operator.*

Validity is preserved by isomorphisms.

In the above definitions we have used a minimal set of combinators; in practice, however, it is convenient to use other, derived, combinators; we list below those that we shall use in this paper, together with their semantics (formal or informal according to which is clearer).

- **true**, **false**, $\vee$, $\wedge$, $\exists$, $\nexists$, $\equiv$, $\exists\, x, y, z \ldots$, $\forall\, x, y, z \ldots$ defined in the usual way
- $\bigtriangledown(t, \pi) =_{\mathrm{def}} \neg \triangle(t, \neg \pi)$
  $DA, V \models \bigtriangledown(t, \pi)$ iff $t^{DA,V}$ is defined and there exists $\sigma \in PATH(DA, ds)$, with ds sort of t, s.t. $S(\sigma) = t^{DA,V}$ and $DA, \sigma, V \models \pi$

– $\Diamond\, \pi =_{\mathrm{def}} \mathbf{true}\, \mathcal{U}\, \pi$ (eventually π)
$DA, \sigma, V \models \Diamond\, \pi$ iff there exists $i > 0$ s.t. $\sigma_{|i}$ is defined and $DA, \sigma_{|i}, V \models \pi$
– $\Box\, \pi =_{\mathrm{def}} \neg \Diamond \neg\, \pi$ (always π)
$DA, \sigma, V \models \Box\, \pi$ iff $DA, \sigma_{|i}, V \models \pi$ for all $i > 0$ s.t. $\sigma_{|i}$ is defined
– $\blacksquare\, \pi =_{\mathrm{def}} \pi \wedge \Box\, \pi$ and $\blacklozenge \pi =_{\mathrm{def}} \pi \vee \Diamond\, \pi$
$\blacksquare$ and $\blacklozenge$ are, respectively, the "always" and "eventually" operators that include the initial (or present) moment
– $\pi_1\, \mathcal{W}\, \pi_2 =_{\mathrm{def}} (\pi_1\, \mathcal{U}\, \pi_2) \vee \Box\, \pi_1$
$\mathcal{W}$ is the weak until operator $\pi_1\, \mathcal{W}\, \pi_2$ holds on a path σ if π_1 holds until π_2 becomes true; the difference with $\mathcal{U}$ is that π_2 may never become true.
– $\bigcirc\, \pi =_{\mathrm{def}} \mathbf{false}\, \mathcal{W}\, \pi$ (next π)
$DA, \sigma, V \models \bigcirc\, \pi$ iff either $\sigma_{|1}$ is not defined or $DA, \sigma_{|1}, V \models \pi$
– $\langle\langle \lambda x \,.\, \phi \rangle\rangle =_{\mathrm{def}} \langle \lambda x \,.\, \phi \rangle \wedge \neg \langle \lambda x \,.\, \mathbf{false} \rangle$
$DA, \sigma, V \models \langle\langle \lambda x \,.\, \phi \rangle\rangle$ iff $L(\sigma)$ is defined and $DA, V[L(\sigma)/x] \models \phi$.
In other words we require that in σ there is an initial transition labelled by $L(\sigma)$ and that this label satisfies ϕ. (Notice that $\langle \lambda x \,.\, \mathbf{false} \rangle$ can be satisfied only by a path which consists of a single state, the initial one.)

Example 11
We give here a few sketchy examples, that should clarify the meaning of the non-standard constructs in our language; in particular, examples c) and d) should explain the role of the binders λx; more significant examples can be found in the following sections. We assume that: Cs is a constant symbol of dynamic sort ds; Ps and Pl are unary predicate symbols of arity ds and $lab(ds)$, respectively; Q is a predicate symbol of arity $ds \times lab(ds) \times ds$; x, x', z, z' and y, y' are variables of sort ds and $lab(ds)$ respectively. Moreover, for simplicity, we do not distinguish between the symbols Cs, Ps, Pl, $\ldots$ and their interpretations.

a) $\triangle(Cs, \Diamond\, \langle \lambda y \,.\, Pl(y) \rangle)$
can be read as: on every infinite path from the state Cs there exists a label that satisfies Pl; (all finite paths trivially satisfy $\Diamond\, \langle \lambda y \,.\, Pl(y) \rangle$).
b) $\bigtriangledown(Cs, \Box\, \Diamond\, [\, \lambda x \,.\, Ps(x)\,])$
can be read as: there exists a path from the state Cs that contains infinitely many states satisfying Ps.
c) $\triangle(Cs, \Box\, [\, \lambda x \,.\, \bigtriangledown(x, \Diamond\, [\, \lambda z \,.\, Ps(z)\,])\,])$ *can be read as: for every path σ from Cs, for every state x on σ, there is a path from x such that along this path there is a state z satisfying Ps.*
d) $\bigtriangledown(Cs, \Box\, \forall\, x, y, z \,.$
$([\, \lambda x' \,.\, x' = x\,] \wedge \langle \lambda y' \,.\, y' = y \rangle \wedge \bigcirc\, [\, \lambda z' \,.\, z' = z\,]) \supset Q(x, y, z))$
can be read as: there exists a path σ from Cs along which any transition $x \xrightarrow{y} z$ is such that $Q(x, y, z)$ is true. A more concise formula expressing the same property is:
$\bigtriangledown(Cs, \Box\, \forall\, x, y \,.\, ([\, \lambda x' \,.\, x' = x\,] \wedge \langle \lambda y' \,.\, y' = y \rangle) \supset \bigcirc\, [\, \lambda z \,.\, Q(x, y, z)\,]).$

Dynamic signatures, dynamic algebras, dynamic homomorphisms and the formulae of our logic form an institution $\mathcal{DYN}$, as defined in [17]. This can be seen by adapting to our case definitions in [3]; however here we assume only that the reader is familiar with the main definitions in [17].

We shall prove that $\mathcal{DYN} = (\mathbf{DSign}, \mathrm{DSen}, \mathrm{DAlg}, \models)$ is an institution, where the four ingredients $\mathbf{DSign}$, DSen, DAlg, $\models$ are defined as follows.

Recall that: $\mathcal{X}$ is a fixed set of variable symbols; if $D\Sigma$ is a signature with set of sorts SRT, a *sort assignment* for $\mathcal{X}$ w.r.t. $D\Sigma$ is a partial function $X\colon \mathcal{X} \to SRT$; moreover we also regard X as an SRT-indexed family $\{X_{srt}\}_{srt \in SRT}$, where $X_{srt} = \{\chi \mid \chi \in \mathcal{X} \text{ and } X(\chi) = srt\}$. We also assume that X_{srt} is infinite and denumerable for every srt. Finally, let **Cat** be a category "sufficiently large" to include, as objects, all categories of algebras.

The signature category Given two (predicate) signatures $\Sigma = (SRT, OP, PR)$ and $\Sigma' = (SRT', OP', PR')$, a *(predicate) signature morphism* $\rho\colon \Sigma \to \Sigma'$ is a triple (τ, φ, ψ) such that:

- $\tau\colon SRT \to SRT'$ is a total map;
- $\varphi\colon OP \to OP'$ is a total map s.t. if $Op\colon s_1 \times \ldots \times s_n \to s$ then $\varphi(Op)\colon \tau(s_1) \times \ldots \times \tau(s_n) \to \tau(s)$;
- $\psi\colon PR \to PR'$ is a total map s.t. if $Pr\colon s_1 \times \ldots \times s_n$ then $\psi(Pr)\colon \tau(s_1) \times \ldots \times \tau(s_n)$.

In what follows we shall confuse τ, φ and ψ, therefore we simply use ρ instead. Given two dynamic signatures $D\Sigma = (\Sigma, DS)$ and $D\Sigma' = (\Sigma', DS')$, a *dynamic signature morphism* $\rho\colon D\Sigma \to D\Sigma$ is a predicate signature morphism $\rho\colon \Sigma \to \Sigma'$ such that:

- $\rho(DS) \subseteq DS'$;
- for all $ds \in DS$: $\rho(lab(ds)) = lab(\rho(ds))$ and
  $\rho(\_ \xrightarrow{\_} \_\colon ds \times lab(ds) \times ds) = \_ \xrightarrow{\_} \_\colon \rho(ds) \times lab(\rho(ds)) \times \rho(ds)$.

It is easy to see that dynamic signatures and dynamic signature morphisms form a category that we call **DSign**.

The sentence functor DSen: **DSign** $\to$ **Set** (**Set** is the category of sets) is the functor defined as follows:

- on objects:
  $\mathrm{DSen}(D\Sigma) =$
  $\cup\{(X, \phi) \mid X$ is a sort assignment for $\mathcal{X}$ w.r.t. $D\Sigma$ and $\phi \in F_{D\Sigma}(X)\}$;
- on morphisms:
  $\mathrm{DSen}(\rho: D\Sigma \to D\Sigma'): \mathrm{DSen}(D\Sigma) \to \mathrm{DSen}(D\Sigma')$ is the mapping sending (X, ϕ) into $(\rho(X), \rho(\phi))$; where $\rho(X)$ is the sort assignment for $\mathcal{X}$ w.r.t. $D\Sigma'$ defined by $\rho(X)(\chi) = \rho(X(\chi))$ and $\rho(\phi)$ is the formula obtained by replacing in ϕ each symbol $sym \in D\Sigma$ with $\rho(sym)$. Notice that in going from ϕ to $\rho(\phi)$ variables do not change; their sorts, however, do change and in the appropriate way (sort assignments have been introduced precisely to this purpose).

It is trivial to see that DSen is a functor.

The algebra functor $\mathrm{DAlg}: \mathbf{DSign} \to \mathbf{Cat}^{\mathbf{OP}}$ ($\mathbf{Cat}^{\mathbf{OP}}$ is the opposite of the category $\mathbf{Cat}$) is the functor defined as follows:

- on objects: $\mathrm{DAlg}(D\Sigma) = \mathbf{DAlg}_{D\Sigma}$ ($\mathbf{DAlg}_{D\Sigma}$ is the category of $D\Sigma$-algebras, see Sect. 3);
- on morphisms: $\mathrm{DAlg}(\rho: D\Sigma \to D\Sigma'): \mathbf{DAlg}_{D\Sigma'} \to \mathbf{DAlg}_{D\Sigma}$ is the mapping sending a $D\Sigma'$ algebra DA' into the $D\Sigma$ algebra DA given by:
  · $DA_{srt} = DA'_{\rho(srt)}$ for all $srt \in SRT$;
  · $Op^{DA} = \rho(Op)^{DA'}$ for all Op in OP and similarly for all Pr in PR.

It is easy to see that DAlg is a functor.

The satisfaction relations For each dynamic signature $D\Sigma$, the satisfaction relation $\models_{D\Sigma}$ is s.t. for each DA in $\mathbf{DAlg}_{D\Sigma}$, and each (X, ϕ) in $\mathrm{DSen}(D\Sigma)$:

$$DA \models_{D\Sigma} (X, \phi) \quad \text{iff} \quad DA \models \phi \quad \text{(see Def. 9)}.$$

Proposition 12 *$\mathcal{DYN}$ is an institution.*

Proof. *All we have to prove is that for each $\rho: D\Sigma \to D\Sigma'$ the* satisfaction condition *holds: for each DA' in $\mathbf{DAlg}_{D\Sigma'}$, for each (X, ϕ) in* $\mathrm{DSen}(D\Sigma)$

$$DA' \models_{D\Sigma'} \mathrm{DSen}(\rho)(X, \phi) \quad \textit{iff} \quad \mathrm{DAlg}(\rho)(DA') \models_{D\Sigma} (X, \phi)$$

i.e.

$$DA' \models \rho(\phi) \quad \textit{iff} \quad DA \models \phi \qquad \textit{where} \quad DA = \mathrm{DAlg}(\rho)(DA').$$

First of all, we need a canonical transformation of variable evaluations. If V' is a variable evaluation $V': \rho(X) \to DA'$, let $\mathrm{DAlg}(\rho)(V')$ *denote the variable* *evaluation* $\mathrm{DAlg}(\rho)(V')\colon X \to DA$ *defined as follows:* $\mathrm{DAlg}(\rho)(V')(\chi) = V'(\chi)$. *Notice that* $\mathrm{DAlg}(\rho)(V')$ *is well-defined, since* $\chi \in X_{srt} \Rightarrow \chi \in \rho(X)_{\rho(srt)} \Rightarrow V'(\chi) \in DA'_{\rho(srt)} = DA_{srt}$. *Then we need the following lemmas, where we use* V *instead of* $\mathrm{DAlg}(\rho)(V')$.

Lemma 13 *For all* $t \in T_{D\Sigma}(X)$*:* $\rho(t)^{DA',V'} = t^{DA,V}$*, where* $\rho(t)$ *is the term obtained by replacing, in* t*, each symbol* $sym \in D\Sigma$ *with* $\rho(sym)$.

Proof. *Straightforward, by induction on* t. $\square$

Lemma 14

(i) $DA', V' \models \rho(\phi)$ *iff* $DA, V \models \phi$;

(ii) $DA', \sigma', V' \models \rho(\pi)$ *iff* $DA, \sigma', V \models \pi$*, when* $\pi \in P_{D\Sigma}(X, ds)$ *and* σ' *is a path on* DA'*; note that the r.h.s. above is well-defined since* $PATH(DA', \rho(ds)) = PATH(DA, ds)$ *and so* σ' *may be used in both sides.*

Proof. *By multiple induction on* ϕ *and* π.

Base.

- $\phi = Pr(t_1, \dots, t_n)$.
  $DA', V' \models \rho(Pr(t_1, \dots, t_n))$ *iff* $DA', V' \models \rho(Pr)(\rho(t_1), \dots, \rho(t_n))$ *iff* $(\rho(t_1)^{DA',V'}, \dots, \rho(t_n)^{DA',V'}) \in \rho(Pr)^{DA'}$ *iff*
  (by Lemma 13 and the definition of DA)
  $(t_1^{DA,V}, \dots, t_n^{DA,V}) \in Pr^{DA}$ *iff* $DA, V \models Pr(t_1, \dots, t_n)$.
- $\phi = t_1 = t_2$. *Analogously to the above case.*

Step. *By cases on the main logical combinator in* ϕ *(or* π*).*

- $\phi = \triangle(t, \pi)$*, with* t *term of sort* ds.
  $DA', V' \models \rho(\triangle(t, \pi))$ *iff* $DA', V' \models \triangle(\rho(t), \rho(\pi))$ *iff*
  $DA', \sigma', V' \models \rho(\pi)$ *for all* $\sigma' \in PATH(DA', \rho(ds))$ *s.t.* $S(\sigma') = \rho(t)^{DA',V'}$
  iff (since $PATH(DA', \rho(ds)) = PATH(DA, ds)$*, by Lemma 13 and by the inductive hypothesis applied to* π*)*
  $DA, \sigma', V \models \pi$ *for all* $\sigma' \in PATH(DA, ds)$ *s.t.* $S(\sigma') = t^{DA,V}$ *iff*
  $DA, V \models \triangle(t, \pi)$.
- $\phi = \phi_1 \supset \phi_2$, $\neg\, \phi_1$, $\forall\, x \,.\, \phi_1$; $\pi = \pi_1 \supset \pi_2$, $\neg\, \pi_1$, $\forall\, x \,.\, \pi_1$. *Easy verifications.*
- $\pi = [\, \lambda x \,.\, \phi \,]$.
  $DA', \sigma', V \models \rho([\, \lambda x \,.\, \phi \,])$ *iff* $DA', V', \sigma' \models [\, \lambda x \,.\, \rho(\phi) \,]$ *iff*
  $DA', V'[S(\sigma')/x] \models \rho(\phi)$ *iff (by the inductive hypothesis)*
  $DA, \mathrm{DAlg}(\rho)(V[S(\sigma')/x]) \models \phi$ *iff* $DA, \mathrm{DAlg}(\rho)(V)[S(\sigma')/x] \models \phi$ *iff*
  $DA, V[S(\sigma')/x] \models \phi$ *iff* $DA, \sigma', V \models [\, \lambda x \,.\, \phi \,]$.
- $\pi = \langle \lambda x \,.\, \phi \rangle$. *Analogously to the above case.*
- $\pi = \pi_1 \,\mathcal{U}\, \pi_2$.
  $DA', \sigma', V' \models \rho(\pi_1 \,\mathcal{U}\, \pi_2)$ *iff* $DA', \sigma', V' \models \rho(\pi_1) \,\mathcal{U}\, \rho(\pi_2)$ *iff*

there exists $j > 0$ *s.t.* $DA', \sigma'_{|j}, V' \models \rho(\pi_2)$ *and*
$DA', \sigma'_{|i}, V' \models \rho(\pi_1)$ *for all* i, $0 < i < j$, *iff*
(by the inductive hypothesis) there exists $j > 0$ *s.t.* $DA, \sigma'_{|j}, V \models \pi_2$ *and*
$DA, \sigma'_{|i}, V \models \pi_1$ *for all* i, $0 < i < j$, *iff* $DA, \sigma', V \models \pi_1 \, \mathcal{U} \, \pi_2$. $\square$

Coming back to the main proof, we show that $DA' \models \rho(\phi)$ *iff* $DA \models \phi$, *thus concluding the proof.*
$DA' \models \rho(\phi)$ *iff (by definition of validity)*
for all $V'\colon \rho(X) \to DA'$, $DA', V' \models \rho(\phi)$ *iff (by Lemma 14)*
for all $V'\colon \rho(X) \to DA'$ $DA, V \models \phi$ *where* $V = \mathrm{DAlg}(\rho)(V')$ *iff*
(since all $V\colon X \to DA$ *can be obtained as* $\mathrm{DAlg}(\rho)(V')$ *for some* $V'\colon \rho(X) \to DA'$*)*
for all $V\colon X \to DA$, $DA, V \models \phi$, *i.e.* $DA \models \phi$. $\square$

4.3 Deductive Systems

4.3.1 Strong incompleteness result

Our logic does not admit a sound and complete effective deductive system. Here "effective" means that: the set of axioms is recursive, proofs are finite objects (hence the system is finitary) and the relation "p is a proof of formula ϕ from the set $\mathcal{A}$ of axioms" is decidable. The set of theorems of an effective deductive system is recursively enumerable, see e.g. [20, Chapt. 8, Sect. 2]. Following [51] we can say that our logic is *strongly incomplete*: the set of valid sentences over *any signature*, provided it contains one dynamic sort (and the relative label sort and transition predicate), is not recursively enumerable.

In [51] the result is proved for a 1st order linear temporal logic with until, where variables can be *global* (*rigid*), as they are in our logic, or *local* (*flexible*): variables that can change their value as time flows. Local variables are an essential ingredient in the incompleteness proofs in [51] (and also in those given in [34] and [1]) and we do not have them in our language. However (as it is done in [51]) we can follow the pattern used to prove the incompleteness of 2nd order logic in [21].

For the rest of this paragraph, let $D\Sigma$ be any dynamic signature containing a dynamic sort, ds. Hence $D\Sigma$ contains also the sort $lab(ds)$ and the predicate symbol $\_ \xrightarrow{\_} \_\colon ds \times lab(ds) \times ds$. Moreover let X be as in Sect. 2.

Lemma 15 *There is a closed formula* $\phi_{\mathrm{fin}} \in F_{D\Sigma}(X)$ *s.t. for any* $D\Sigma$*-algebra* DA, *if* $DA \models \phi_{\mathrm{fin}}$ *then* DA_{ds} *is a finite set.*

Proof. $\phi_{\mathrm{fin}} =_{\mathrm{def}} \mathrm{EL} \wedge \mathrm{UL} \wedge \mathrm{FUN} \wedge \mathrm{REACH} \wedge \mathrm{FIN}$, *where assuming that x, x' are variables of sort $lab(ds)$ and y, y', z, z' are variables of sort ds:*

$\mathrm{EL} =_{\mathrm{def}} \exists\, x \,.\, x = x$ *(existence of labels)*

$\mathrm{UL} =_{\mathrm{def}} \forall\, x, x' \,.\, x = x'$ *(uniqueness of labels)*

$\mathrm{FUN} =_{\mathrm{def}} \forall\, y, z, z', x \,.\, ((y \xrightarrow{x} z \ \wedge\ y \xrightarrow{x} z') \supset z = z')$
($\xrightarrow{x}$ is a function)

$\mathrm{REACH} =_{\mathrm{def}} \exists\, y \,.\, \forall\, z \,.\ \bigtriangledown (y, \blacklozenge[\,\lambda z' \,.\, z' = z\,])$
(every element can be reached using $\rightarrow^*$ from a single one)

$\mathrm{FIN} =_{\mathrm{def}} \forall\, y \,.\ \bigtriangleup (y, \blacklozenge\langle \lambda x \,.\, \mathbf{false}\rangle)$
(every path is finite; recall that $\langle \lambda x \,.\, \mathbf{false}\rangle$ is satisfied at the end point of a path, because there is no "next label").

EL $\wedge$ UL *simplifies the picture: it requires the existence of a unique label of sort $lab(ds)$ (uniqueness is needed later);* FUN $\wedge$ REACH *implies that the elements of sort ds can be arranged along a (unique) non-empty sequence and* FIN *forces it to be finite, allowing for just a finite number of elements of sort ds.* $\square$

As an aside we also show that we can characterize (up to bijections) the set of natural numbers.

Let $\phi_{\mathrm{nat}} =_{\mathrm{def}} \mathrm{EL} \wedge \mathrm{UL} \wedge \mathrm{FUN} \wedge \mathrm{REACH} \wedge \mathrm{DIFF} \wedge \mathrm{INF}$,
where EL, UL, FUN and REACH are as above and

$\mathrm{DIFF} =_{\mathrm{def}} \forall\, y \,.\ \bigtriangleup (y, \Box\,[\,\lambda z \,.\, z \neq y\,])$

$\mathrm{INF} =_{\mathrm{def}} \forall\, y \,.\, \exists\, y', x \,.\, y \xrightarrow{x} y'$

EL $\wedge$ UL $\wedge$ FUN $\wedge$ REACH requires that all the elements of sort ds can be arranged in a unique non-empty sequence. Then DIFF forces all elements to be different and INF forces the sequence to be infinite. Therefore: if $DA \models \phi_{\mathrm{nat}}$, then $DA_{lab(ds)}$ is a singleton set and DA_{ds} is in bijection with the set of the natural numbers.

Coming back to the main issue in this paragraph, we show that, in our logic, validity is not even semi-decidable.

Theorem 16 *The set $\mathcal{VS}$ of closed formulae in $F_{D\Sigma}(X)$ that are valid in every $D\Sigma$-algebra is not recursively enumerable.*

Proof. *Let x be a (fixed) variable of sort $lab(ds)$ and F^{ds} be the set of sentences (closed formulae) of our logic built using only: variables of sort ds, $=$, $\neg$, $\supset$, $\forall$ and the* binary *predicate symbol* $\xrightarrow{x} : ds \times ds$. *Moreover let*

$$F^{ds}_{\text{fin}} = \{\phi \mid \phi \in F^{ds} \textit{ and } \models \phi_{\text{fin}} \supset \phi\},$$

where ϕ_{fin} is the formula exhibited in Lemma 15.

F^{ds}_{fin} is therefore isomorphic to the set of classical *1st order sentences built using one binary predicate symbol which are valid in all structures with finite (non-empty) domain. This last set is not recursively enumerable ([21, pg. 164, bottom lines]).*
To show that $\mathcal{VS}$ is not recursively enumerable it suffices to show that we can reduce F^{ds}_{fin} to $\mathcal{VS}$. This is obtained using the function $\rho: F_{D\Sigma}(X) \to F_{D\Sigma}(X)$
$$s.t.\ \rho(\phi) = \begin{cases} \textbf{false} \textit{ if } \phi \notin F^{ds} \\ \phi_{\text{fin}} \supset \phi \textit{ if } \phi \in F^{ds} \end{cases} \quad \textit{(indeed } \phi \in F^{ds}_{\text{fin}} \textit{ iff } (\phi_{\text{fin}} \supset \phi) \in \mathcal{VS}\textit{).} \quad \Box$$

As an immediate corollary of this theorem we have the incompleteness result: our logic does not admit a complete effective deductive system.

To overcome this problem we could look for complete *infinitary* systems (but we have not tried yet) or consider interesting fragments of our logic that are more manageable. In subsection 4.4.1 we shall present a complete (finitary) system for "dynamic positive conditional formulae". A third way out is discussed in the next paragraph.

4.3.2 A sound deductive system for our logic

Even if completeness cannot be achieved, it is worthwhile to have a (sound) system where the rules express significant basic properties of the various combinators. First of all we must extend some standard definitions.

Free and bound occurrences of variables in state and path formulae are defined as usual by adding the clauses:

- $\mathcal{U}$ and all its derived temporal connectives behave like the propositional connectives;
- all occurrences of x in t are free in $\triangle(t, \pi)$;
- all free/bound occurrences of x in π are free/bound occurrences of x in $\triangle(t, \pi)$;
- all occurrences of x in $[\,\lambda x \,.\, \phi\,]$ and $\langle \lambda x \,.\, \phi \rangle$ are bound;
- if $x \neq y$, all free/bound occurrences of x in ϕ are free/bound occurrences of x in $[\,\lambda y \,.\, \phi\,]$ and $\langle \lambda y \,.\, \phi \rangle$.

Then substitution is extended in the obvious way; for instance:

$$\triangle(t, [\,\lambda x \,.\, \phi\,]\, \mathcal{U}\, \langle \lambda y \,.\, \phi\rangle)[t'/x] \qquad \text{is} \qquad \triangle\,(t[t'/x], [\,\lambda x \,.\, \phi\,]\, \mathcal{U}\, \langle \lambda y \,.\, \phi[t'/x]\rangle);$$

notice that here x and y must be different because they have different sorts. When writing "$\ldots[t/x]$" we shall assume that substitution is legal: sorts are respected and there is no capture of free occurrences of variables.

If ϕ' is obtained from ϕ by consistently changing the names of bound variables then we say that ϕ' is obtained from ϕ by α *conversion.*

A formula is *tautologically valid* iff it is obtained from a propositional tautology by consistently replacing propositional variables with formulae. It is straightforward to verify that tautologically valid formulae are valid in any algebra.

ϕ is a *tautological consequence* of $\phi_1, \ldots, \phi_n$ $(n \geq 1)$ iff $(\phi_1 \,\wedge\, \ldots \,\wedge\, \phi_n) \,\supset\, \phi$ is tautologically valid. We use $\{\phi_1, \ldots, \phi_n\} \vdash_\tau \phi$ to denote tautological consequence $(n \geq 1)$ or tautological validity $(n = 0)$. Analogous definitions apply to path formulae.

As dynamic formulae are the main ones in our language, the system below, $\mathcal{DS}$, refers to them. However some rules express properties of path formulae (indeed $\triangle(x, \pi)$ holds iff π holds on every path).

There are three groups of rules. The first one defines a sound and complete system for 1st order logic with partial functions; its core is in [52] Chapter 2, Section 2. Notice that axiom $[\alpha]$ is useful in connections with axioms like $[\triangle\forall]$. In the second group we have axioms that "define" the combinators $\langle\lambda x \,.\, \ldots\rangle$, $[\,\lambda x \,.\, \ldots\,]$ and (recursively) $\mathcal{U}$, together with the strictness property for $\triangle$. The last group consists of axioms and rules describing the interaction of the temporal combinators with propositional connectives and quantification.

We recall that $D(t)$ stands for $t = t$.

1st order

[ref] $\quad x = x$

[rep] $\quad t = t' \,\supset\, (\phi[t/x] \,\supset\, \phi[t'/x])$

[str1] $\quad D(Op(t_1, \ldots, t_n)) \,\supset\, D(t_i)\ ,\ 1 \leq i \leq n$

[str2] $\quad Pr(t_1, \ldots, t_n) \,\supset\, D(t_i)\ ,\ 1 \leq i \leq n$

[part] $\quad (\forall\, x \,.\, \phi) \,\supset\, (D(t) \,\supset\, \phi[t/x])$

[$\forall \supset$] $(\forall\, x\,.\,\phi \supset \phi') \supset (\phi \supset \forall\, x\,.\,\phi')$, where x does not appear free in ϕ

[gen] $\dfrac{\phi}{\forall\, x\,.\,\phi}$

[taut] $\dfrac{\phi_1 \ldots\ \phi_n}{\phi}$ where $\{\phi_1, \ldots, \phi_n\} \vdash_{\tau} \phi$, $n \geq 0$

[α] $\dfrac{\phi}{\phi'}$ where ϕ' is obtained from ϕ by α-conversion

Definition of the temporal connectives

[str3] $\triangle(t, \pi) \supset D(t)$

[D1] $\triangle(t, [\,\lambda x\,.\,\phi\,]) \equiv (D(t) \wedge \phi[t/x])$

[D2] $\triangle(t, \langle \lambda x\,.\,\phi \rangle) \equiv (D(t) \wedge (\forall\, y, z\,.\, t \xrightarrow{y} z \supset \phi[y/x]))$

[D3] $\triangle(t, \pi\, \mathcal{U}\, \pi') \equiv$
$D(t) \wedge (\exists\, y, z\,.\, t \xrightarrow{y} z) \wedge (\forall\, y, z\,.\, t \xrightarrow{y} z \supset \triangle(z, \pi' \vee (\pi \wedge \pi\, \mathcal{U}\, \pi')))$

in [D2] and [D3]: z and y are “new” variables

[$\Box$ Gen] $\dfrac{\triangle(x, \pi)}{\triangle(x, \Box\, \pi)}$

Interaction between temporal and 1$^{\text{st}}$ order connectives

[$\triangle\neg$] $\triangle(t, \neg\, \pi) \supset \neg\, \triangle(t, \pi)$

[$\triangle \supset$] $\triangle(t, \pi \supset \pi') \supset (\triangle(t, \pi) \supset \triangle(t, \pi'))$

[$\triangle\forall$] $\triangle(t, \forall\, x\,.\,\pi) \equiv \forall\, x\,.\, \triangle(t, \pi)$, where x does not appear in t

[L$\neg$] $\triangle(t, \langle \lambda x\,.\,\neg\, \phi \rangle) \equiv \triangle(t, \neg\, \langle \lambda x\,.\,\phi \rangle)$

[L$\supset$] $\triangle(t, \langle \lambda x\,.\,\phi \supset \phi' \rangle) \equiv \triangle(t, \langle \lambda x\,.\,\phi \rangle \supset \langle \lambda x\,.\,\phi' \rangle)$

[L$\forall$] $\triangle(t, \langle \lambda x\,.\,\forall\, y\,.\,\phi \rangle) \equiv \triangle(t, \forall\, y\,.\,\langle \lambda x\,.\,\phi \rangle)$, where $x \neq y$

[S$\neg$], [S$\supset$], [S$\forall$] obtained from [L$\neg$], [L$\supset$], [L$\forall$] respectively by changing $\langle\ \rangle$ into [ ]

[$\Box$ $\supset$] $\triangle(t, \Box\, (\pi_1 \supset \pi_2)) \supset \triangle(t, (\Box\, \pi_1) \supset \Box\, \pi_2)$

[πtaut] $(\triangle(t, \pi_1) \wedge \ldots \wedge \triangle(t, \pi_n)) \supset \triangle(t, \pi)$, where $\{\pi_1, \ldots, \pi_n\} \vdash_\tau \pi$, $n \geq 0$

[πpart] $(D(t) \wedge D(t')) \supset \triangle(t, (\forall x . \pi) \supset \pi[t'/x])$

[$\pi\forall \supset$] $D(t) \supset \triangle(t, (\forall x . \pi \supset \pi') \supset (\pi \supset \forall x . \pi'))$,
where x is not free in π

[$\mathcal{U}\,\forall$] $\triangle(t, (\forall x . \pi)\, \mathcal{U}\, \pi') \equiv \triangle(t, \forall x . \pi\, \mathcal{U}\, \pi')$,
where x is not free in π'

[$\mathcal{U}$ taut] $(\triangle(t, \pi_i\, \mathcal{U}\, \pi) \wedge \ldots \wedge \triangle(t, \pi_n\, \mathcal{U}\, \pi)) \supset \triangle(t, \pi_0\, \mathcal{U}\, \pi)$,
where $\{\pi_1, \ldots, \pi_n\} \vdash_\tau \pi_0$ and $n \geq 0$.

Theorem 17 *The deductive system $\mathcal{DS}$ is sound: when Γ is a set of dynamic formulae, if $\Gamma \vdash_{\mathcal{DS}} \phi$, then ϕ holds in all the models of Γ.*

Proof. *It is straightforward to verify that for each of the above rules: the consequence holds in an algebra DA whenever all the premises hold in DA.* □

Example 18 Proofs in $\mathcal{DS}$
As an example, we show that in $\mathcal{DS}$ we can prove:

$$\phi(x) \supset \psi(x) \quad \vdash \quad \triangle(y, \blacksquare[\, \lambda x . \phi(x)\,]) \supset \triangle(y, \blacksquare[\, \lambda x . \psi(x)\,])$$

(recall that $\blacksquare\, \pi$ stands for $(\square\, \pi) \wedge \pi$).

(1)	$\phi(x) \supset \psi(x)$	hypothesis
(2)	$\phi(y) \supset \psi(y)$	straightforward, from (1) and the 1st order rules
(3)	$D(y)$	by [Ref]
(4)	$D(y) \wedge (\phi(y) \supset \psi(y))$	by (2), (3) and [taut]
(5)	$(D(y) \wedge (\phi(y) \supset \psi(y))) \supset \triangle(y, [\, \lambda x . \phi(x) \supset \psi(x)\,])$	by [D1]
(6)	$\triangle(y, [\, \lambda x . \phi(x) \supset \psi(x)\,])$	by (4), (5) and [taut]
(7)	$\triangle(y, [\, \lambda x . \phi(x) \supset \psi(x)\,]) \supset \triangle(y, [\, \lambda x . \phi(x)\,] \supset [\, \lambda x . \psi(x)\,])$	by [S$\supset$]
(8)	$\triangle(y, [\, \lambda x . \phi(x)\,] \supset [\, \lambda x . \psi(x)\,])$	by (7), (6) and [taut]
(9)	$\triangle(y, \square\, ([\, \lambda x . \phi(x)\,] \supset [\, \lambda x . \psi(x)\,]))$	by (8) and [$\square$ Gen]
(10)	$\triangle(y, (\square\, [\, \lambda x . \phi(x)\,]) \supset (\square\, [\, \lambda x . \psi(x)\,]))$	by (9) and [$\square \supset$]
(11)	$\triangle(y, (\square\, [\, \lambda x . \phi(x)\,]) \supset \square\, [\, \lambda x . \psi(x)\,]) \supset \triangle(y, \square\, [\, \lambda x . \phi(x)\,]) \supset \triangle(y, \square\, [\, \lambda x . \psi(x)\,])$	by [$\triangle \supset$]
(12)	$\triangle(y, \square\, [\, \lambda x . \phi(x)\,]) \supset \triangle(y, \square\, [\, \lambda x . \psi(x)\,])$	by (10), (11) and [taut]

In a similar way we obtain (from hypothesis (1)):

(13) $\triangle(y, [\, \lambda x . \phi(x)\,]) \supset \triangle(y, [\, \lambda x . \psi(x)\,])$

Then the proof goes on as follows.

(14) $\triangle(y,[\,\lambda x\,.\,\phi(x)\,] \,\wedge\, \Box\,[\,\lambda x\,.\,\phi(x)\,]) \supset \triangle(y,[\,\lambda x\,.\,\phi(x)\,])$ by [πtaut]

(15) $\triangle(y,[\,\lambda x\,.\,\phi(x)\,] \,\wedge\, \Box\,[\,\lambda x\,.\,\phi(x)\,]) \supset \triangle(y,[\,\lambda x\,.\,\psi(x)\,])$
by (13), (14) and [taut]

(16) $\triangle(y,[\,\lambda x\,.\,\phi(x)\,]) \,\wedge\, \triangle(y,\Box\,[\,\lambda x\,.\,\phi(x)\,]) \supset \triangle(y,\Box\,[\,\lambda x\,.\,\phi(x)\,])$
by [πtaut]

(17) $\triangle(y,[\,\lambda x\,.\,\phi(x)\,] \,\wedge\, \Box\,[\,\lambda x\,.\,\phi(x)\,]) \supset \triangle(y,\Box\,[\,\lambda x\,.\,\psi(x)\,])$
by (12), (16) and [taut]

(18) $\triangle(y,[\,\lambda x\,.\,\phi(x)\,] \,\wedge\, \Box\,[\,\lambda x\,.\,\phi(x)\,]) \supset$
$\triangle(y,[\,\lambda x\,.\,\psi(x)\,]) \,\wedge\, \triangle(y,\Box\,[\,\lambda x\,.\,\psi(x)\,])$ by (15), (17) and [taut]

(19) $\triangle(y,[\,\lambda x\,.\,\psi(x)\,]) \,\wedge\, \triangle(y,\Box\,[\,\lambda x\,.\,\psi(x)\,]) \supset$
$\triangle(y,[\,\lambda x\,.\,\psi(x)\,] \,\wedge\, \Box\,[\,\lambda x\,.\,\psi(x)\,])$ by [πtaut]

(20) $\triangle(y,[\,\lambda x\,.\,\phi(x)\,] \,\wedge\, \Box\,[\,\lambda x\,.\,\phi(x)\,]) \supset$
$\triangle(y,[\,\lambda x\,.\,\psi(x)\,] \,\wedge\, \Box\,[\,\lambda x\,.\,\psi(x)\,])$ by (18), (19) and [taut]

4.4 (Simple) Dynamic Specifications

Definition 19 *A* (simple) dynamic specification *is a pair $(D\Sigma, AX)$, where $AX \subseteq F_{D\Sigma}(X)$ and X is an appropriate sort assignment (see Sect. 2).*

Notation: usually the dynamic specification $(D\Sigma, AX)$ will be written as: $D\Sigma$ **axiom** AX.

In the three examples below we use the signature BUFΣ defined in Ex. 3; in Ex. 20 we refer to the initial semantics and in Ex. 21 and 22 to the loose one.

Example 20 Unbounded buffers with a LIFO policy
$\mathrm{BUF}_1 = (\mathrm{BUF}\Sigma, \mathit{BUF\_AX}_1)$ *where $\mathit{BUF\_AX}_1$ consists of the following axioms:*

- *properties of the data contained in the buffers (the terms* 0 *and $Succ(n)$ are always defined):*
  $D(0)$ $\quad D(Succ(n))$
- *static properties (LIFO organization of the buffers)*
  $Get(Put(n,b)) = n$ $\quad Remove(Put(n,b)) = b$
- *definition of the dynamic activity of the buffers*
  $D(Get(b)) \supset b \xrightarrow{O(Get(b))} Remove(b)$
  a buffer can always return its first element (if it exists)
  $b \xrightarrow{I(n)} Put(n,b)$
  a buffer can always receive a value and put it on its stack.

This specification admits initial models (see Prop. 24 below): algebras where the carrier of sort buf is the set of unbounded stacks of natural numbers. Notice that, given our interpretation of =, the axiom $Get(Put(n,b)) = n$

implies $D(Get(Put(n,b)))$, hence also $D(Put(n,b))$, while $Get(Empty)$ and $Remove(Empty)$ are undefined.

Example 21
A very abstract specification of buffers containing natural values; we only require the essential properties; clearly this is a "loose specification".

$\mathrm{BUF}_2 = (\mathrm{BUF}\Sigma, BUF\_AX_2)$ *where BUF_AX_2 consists of the following axioms:*

- *properties of the data contained in the buffers:*
  $D(0) \quad D(Succ(n)) \quad \neg\, 0 = Succ(n) \quad Succ(n) = Succ(m) \supset n = m$
- *static properties (the operations Get and Remove are not defined on the empty buffer):*
  $\neg\, D(Get(Empty)) \qquad \neg\, D(Remove(Empty))$
- *dynamic properties (safety properties):*
  $b \xrightarrow{O(n)} b' \supset n = Get(b) \wedge b' = Remove(b)$
  (specifies the action of returning a value)
  $b \xrightarrow{I(n)} b' \supset b' = Put(b,n)$ *(specifies the action of receiving a value)*

Here we do not give the definition of the operations Get, Put and Remove, as we did in BUF_1*, but we only specify some of their properties; clearly such a specification is oriented to a loose semantics. If DA is an algebra which is a model of* BUF_2*, the set DA_{buf} may contain, for instance, unbounded or bounded buffers, FIFO or LIFO buffers, but also buffers which "loose" any value that they receive.*

Example 22
A very abstract specification of buffers containing natural values where the received values are always returned. BUF_3 *is obtained by adding the axioms below to* BUF_2.

(1) $b \xrightarrow{I(n)} b' \supset \triangle(b', \langle \lambda y \,.\, \neg\, y = I(n)\rangle\ \mathcal{W}\ \langle \lambda y \,.\, y = O(n)\rangle)$
This is a safety property: a buffer that has received a value n must return it before it can receive another copy of n; this ensures that the buffer contains distinct values.

(2) $b \xrightarrow{I(n)} b' \supset \triangle(b', \Diamond\, \langle\langle \lambda y \,.\, y = O(n)\rangle\rangle)$
This is a liveness property: eventually, a received value will be returned; recall that the elements in a buffer are distinct, so we know that it is the same n which appears in $I(n)$ and $O(n)$.

If the buffer interacts with its users in a synchronous way, the last axiom is not very appropriate: indeed in case no one wants to accept the returned value, *this axiom prescribes that the whole system, including the buffer and its users, will eventually deadlock. The problem can be avoided by replacing this last axiom with the two formulae below. They just require that a buffer will have the capability of returning any value it receives (2i) and that such capability remains until the value is actually returned (2ii).*

(2i) $\quad b \xrightarrow{I(n)} b' \supset \triangle(b', \Diamond\, [\, \lambda b'' \,.\, Out\_Cap(b'', n)\,])$

(2ii) $\quad Out\_Cap(b, n) \supset \triangle(b, [\, \lambda b' \,.\, Out\_Cap(b', n)\,]\ \mathcal{W}\ \langle\langle \lambda y \,.\, y = O(n)\rangle\rangle)$

where $Out\_Cap(x, y)$ *stands for* $\exists\, z \,.\, x \xrightarrow{O(y)} z$.

BUF_3*, where we use (2i) and (2ii), specifies a subclass of the buffers defined by* BUF_2*: the buffers where received values are returned (if someone requests them). It includes bounded FIFO buffers, but also, say, unbounded LIFO buffers where the "fair behaviour" is obtained by using auxiliary structures. On the other hand, the initial models of* BUF_1 *are not included, since there each buffer admits paths where condition (2i) is violated.*

4.4.1 Initial models

Not all dynamic specifications admit initial models. Classical (static) specifications are a particular case and it is well known that axioms like
$t_1 = t_2 \vee t_3 = t_4$ or $\exists\, x \,.\, Pr(x)$
do not allow initial models, because none of their models can satisfy the properties in Prop. 1. Using Prop. 6, one can show that the same happens with formulae including existential temporal operators; for instance: $\triangledown(t, \pi)$ or $\triangle(t, \Diamond\, \pi)$. However, as in the case of classical specifications, we can guarantee the existence of initial models by restricting the form of the axioms.

An *atom* is any formula which is either $t_1 = t_2$ or $Pr(t_1, \ldots, t_n)$. A formula $\phi \in F_{D\Sigma}(X)$ is *dynamic positive conditional* iff it has the form
$\bigwedge_{i=1,\ldots,n} \alpha_i \supset \psi$,
where: we assume that $\wedge$ binds tighter than $\supset$, $n \geq 0$, α_i is an atom and ψ is either an atom or has the form $\triangle(t, \pi)$ with π built using $[\ldots]$, $\langle\ldots\rangle$, $\Box$, $\bigcirc$ only, and the formulae inside $[\ldots]$ and $\langle\ldots\rangle$ are themselves dynamic positive conditional. Usual *positive conditional formulae*, see Sect. 3, are a particular case: when ψ is an atom. Sometimes it is convenient to write dynamic positive conditional formulae in clausal-form, then we shall use the sequent $\{\alpha_1, \ldots, \alpha_n\} \Rightarrow \psi$ instead of $\bigwedge_{i=1,\ldots,n} \alpha_i \supset \psi$.

The properties that can be specified using axioms of this kind include "usual" static properties, described using ordinary positive conditional formulae, and many *safety properties* (see the examples above).

Let $dsp = (D\Sigma, AX)$ be a dynamic specification s.t. the formulae in AX are dynamic positive conditional. We are going to show that dsp admits initial models. To this end, consider for each dynamic sort ds of $D\Sigma$ a (new) predicate symbol $Trans_{ds}: ds \times ds$ and the set of (positive conditional) axioms

$$TRANS = \cup\{ TRANS_{ds} \mid ds \in DS\},$$

where, if x, z, w are variables of sort ds and y is a variable of sort $lab(ds)$:

$TRANS_{ds} =$
$\{x \xrightarrow{y} z \supset Trans_{ds}(x, z) \ ; \ Trans_{ds}(x, z) \ \wedge \ Trans_{ds}(z, w) \supset Trans_{ds}(x, w)\}.$

Using the predicates $Trans_{ds}$ we can translate dynamic positive conditional formulae into standard many sorted 1st order logic. If ϕ is a dynamic positive conditional formula, its translation, $T(\phi)$, is defined inductively as follows.

$T(\alpha) = \alpha$

$T(\wedge_{i=1,\dots,n} \alpha_i \supset \psi) = \wedge_{i=1,\dots,n} \alpha_i \supset T(\psi)$

$T(\triangle(t, [\lambda x \,.\, \phi'])) = D(t) \ \wedge \ T(\phi')[t/x]$

$T(\triangle(t, \langle \lambda x \,.\, \phi' \rangle)) = D(t) \ \wedge \ (\forall\, y, z \,.\, t \xrightarrow{y} z \supset T(\phi')[y/x])$
y, z not appearing in $T(\phi')$ and t;

$T(\triangle(t, \Box\, \pi)) = D(t) \ \wedge \ (\forall\, z \,.\, Trans(t, z) \supset T(\triangle(z, \pi)))$
z not appearing in π and t;

$T(\triangle(t, \bigcirc\, \pi)) = D(t) \ \wedge \ (\forall\, y, z \,.\, t \xrightarrow{y} z \supset T(\triangle(z, \pi)))$
z not appearing in π and t; y not appearing in t and $T(\triangle(z, \pi))$.

Finally, let dsp' be $(D\Sigma', AX')$ where: $D\Sigma'$ is the signature obtained by adding the predicates $Trans_{ds}$ to $D\Sigma$ and AX' is the set of axioms
$TRANS \cup \{T(\phi) \mid \phi \in AX\}$.

We have the following lemma:

Lemma 23 *For every $D\Sigma'$-algebra DA in $Mod(dsp')$, every variable evaluation V and every dynamic positive conditional formula $\phi \in F_{D\Sigma}(X)$:*
$DA, V \models T(\phi)$ *iff* $DA_{|D\Sigma}, V \models \phi$.

Proof. *Assume $\phi = \wedge_{i=1,\dots,n} \alpha_i \supset \psi$; then we use induction on the maximum depth of nesting of the temporal combinators $\triangle(\_, \_)$, $\Box$ and $\bigcirc$ in ψ.*

Basis: $\psi = \alpha_0$.
Obvious, since for $j = 0, \dots, n$: $DA, V \models \alpha_j$ iff $DA_{|D\Sigma}, V \models \alpha_j$.

Step: *We proceed by cases, according to the shape of ψ.*

- $\psi = \triangle(t, \Box\, \pi)$.
  We have shown in the basis that:
  $DA_{|D\Sigma}, V \models \wedge_{i=1,\ldots,n}\, \alpha_i$ *iff* $DA, V \models \wedge_{i=1,\ldots,n}\, \alpha_i$.
  As $T(\wedge_{i=1,\ldots,n}\, \alpha_i \supset \triangle(t, \Box\, \pi)) = \wedge_{i=1,\ldots,n}\, \alpha_i \supset T(\triangle(t, \Box\, \pi))$*, all we have to show is:*
  $DA_{|D\Sigma}, V \models \triangle(t, \Box\, \pi)$ *iff* $DA, V \models T(\triangle(t, \Box\, \pi))$.
  $DA_{|D\Sigma}, V \models \triangle(t, \Box\, \pi)$ *iff* t *has a defined value, say* s*, in* $DA_{|D\Sigma}$ *w.r.t.* V *and* $DA_{|D\Sigma}, \sigma, V \models \Box\, \pi$ *for all paths* σ *from* s*. As* t *is on the signature* $D\Sigma$*,* $t^{DA,V} = s$*, hence* $DA, V \models D(t)$*. Moreover:*
  $DA_{|D\Sigma}, \sigma, V \models \Box\, \pi$ *for all paths* σ *from* s *iff*
  $DA_{|D\Sigma}, \sigma_{|i}, V \models \pi$ *for all paths* σ *from* s *and all* $i > 0$ *iff*
  $DA_{|D\Sigma}, \sigma', V \models \pi$ *for all paths* σ' *from* s'*, with* s' *s.t. there exists a path from* s *to* s' *iff* $DA, V \models \forall\, z\, .\, Trans(t,z) \supset \triangle(z, \pi)$ *iff*
  $DA, V' \models Trans(t,z) \supset \triangle(z,\pi)$ *for every* V' *which coincides with* V *everywhere but in* z.
  Now: $DA, V' \models \triangle(z,\pi)$ *iff*
  $DA_{|D\Sigma}, V' \models \triangle(z,\pi)$*, as* $\triangle(z,\pi)$ *contains only symbols in* $D\Sigma$*, iff*
  $DA, V' \models T(\triangle(z,\pi))$*, by inductive hypothesis on* $\triangle(z,\pi)$.
  Therefore, (using the properties of $\models$ *and* $\supset$*):*
  $DA, V' \models Trans(t,z) \supset \triangle(z,\pi)$ *for every* V' *which coincides with* V *everywhere but in* z *iff*
  $DA, V' \models Trans(t,z) \supset T(\triangle(z,\pi))$ *for every* V' *which coincides with* V *everywhere but in* z *iff*
  $DA, V \models \forall\, z\, .\, Trans(t,z) \supset T(\triangle(z,\pi))$.
  Recalling that $DA, V \models D(t)$*, we thus obtain:*
  $DA_{|D\Sigma}, V \models \triangle(t, \Box\, \pi)$ *iff* $DA, V \models T(\triangle(t, \Box\, \pi)$*, as required.*
- $\psi = \triangle(t, \bigcirc\, \pi)$.
  Similar to the case above.
- $\psi = \triangle(t, [\lambda x\, .\, \ldots])$, $\psi = \triangle(t, \langle\lambda x\, .\, \ldots\rangle)$.
  Straightforward. $\Box$

The translation of a dynamic positive conditional ϕ is not a positive conditional formula, in general; however, clearly, it is equivalent to a set of positive conditional formulae. We think an example should suffice and omit the formal definition. If ϕ is

$$\wedge_{i=1,\ldots,n}\, \alpha_i \supset \triangle(t, \langle\lambda x\, .\, \phi'\rangle),$$

then $T(\phi)$ is

$$\wedge_{i=1,\ldots,n}\, \alpha_i \supset (D(t) \wedge (\forall\, y, z\, .\, t \xrightarrow{y} z \supset T(\phi')[y/x]))$$

which is equivalent to the set

$$\{\wedge_{i=1,\ldots,n}\, \alpha_i \supset D(t),\ (\wedge_{i=1,\ldots,n}\, \alpha_i \wedge t \xrightarrow{y} z) \supset T(\phi')[y/x]\}$$

i.e. $\{\{\alpha_1, \ldots, \alpha_n\} \Rightarrow D(t), \{\alpha_1, \ldots, \alpha_n, t \xrightarrow{y} z\} \Rightarrow T(\phi')[y/x]\}$.

Proposition 24 *Let $dsp = (D\Sigma, AX)$ be a dynamic specification; if the formulae in AX are dynamic positive conditional, then $Mod(dsp)$ has initial models.*

Proof. *Consider $dsp' = (D\Sigma', AX')$ as defined above. As we have seen, each axiom in AX' is equivalent to a set of positive conditional axioms and thus there exists I initial in $Mod(dsp')$. By Lemma 23, $Mod(dsp) = \{DA_{|D\Sigma} \mid DA \in Mod(dsp')\}$; thus $I_{|D\Sigma}$ is initial in $Mod(dsp)$.* $\square$

If the axioms of *dsp* are dynamic positive conditional formulae, we also have a deductive system which is sound and complete for dynamic positive conditional formulae with respect to *Mod*(*dsp*). The first step is to consider a deductive system, $\mathcal{P}$, for 1st order positive conditional formulae with partially defined terms (but recall that algebras have nonempty carriers). This can be obtained, specializing to our case a system in [41]. In the rules below: Γ is a finite set of atoms; α, β, possibly with subscripts, are atoms; Γ, α abbreviates $\Gamma \cup \{\alpha\}$; $\Gamma[t/x]$ means substitution applied to all the atoms in Γ.

$\mathcal{P}$ is given by the following axioms and rules:

[base] $\alpha \Rightarrow \alpha$ [aug] $\dfrac{\Gamma \Rightarrow \alpha}{\Gamma, \beta \Rightarrow \alpha}$

[ref] $\Rightarrow x = x$ [sym] $\dfrac{\Gamma \Rightarrow t = t'}{\Gamma \Rightarrow t' = t}$ [tr] $\dfrac{\Gamma \Rightarrow t = t' \quad \Gamma \Rightarrow t' = t''}{\Gamma \Rightarrow t = t''}$

[cong1] $$\frac{\Gamma \Rightarrow D(Op(t_1, \ldots, t_n)) \quad \Gamma \Rightarrow t_i = t'_i \;\; i = 1, \ldots, n}{\Gamma \Rightarrow Op(t_1, \ldots, t_n) = Op(t'_1, \ldots, t'_n)}$$

[cong2] $$\frac{\Gamma \Rightarrow Pr(t_1, \ldots, t_n) \quad \Gamma \Rightarrow t_i = t'_i \;\; i = 1, \ldots, n}{\Gamma \Rightarrow Pr(t'_1, \ldots, t'_n)}$$

[str1] $$\frac{\Gamma \Rightarrow D(Op(t_1, \ldots, t_n))}{\Gamma \Rightarrow D(t_i)} \quad 1 \leq i \leq n$$

[str2] $$\frac{\Gamma \Rightarrow Pr(t_1, \ldots, t_n)}{\Gamma \Rightarrow D(t_i)} \quad 1 \leq i \leq n$$

[sub] $\dfrac{\Gamma \Rightarrow \alpha \quad \Gamma \Rightarrow D(t)}{\Gamma[t/x] \Rightarrow \alpha[t/x]}$ [cut] $\dfrac{\Gamma, \beta \Rightarrow \alpha \quad \Gamma' \Rightarrow \beta}{\Gamma \cup \Gamma' \Rightarrow \alpha}$

$\mathcal{P}$ is a particular instance (obtained by using the axioms for equality) of a "parametric" deductive system for 1st order logic that is sound and complete w.r.t. the intuitionistic semantics ([41, sect. 3.2]). On the other hand (see e.g. [41, sect. 3.1]) the intuitionistic semantics and the classical one coincide when we restrict ourselves to positive conditional formulae. Therefore $\mathcal{P}$ is sound and complete: if AX is a set of positive conditional axioms (that we regard as written in clausal form when used in connection with $\mathcal{P}$) then

$$Mod(AX) \models \alpha_1 \wedge \ldots \wedge \alpha_n \supset \alpha_0 \qquad \text{iff} \qquad AX \vdash^{\mathcal{P}} \alpha_1, \ldots, \alpha_n \Rightarrow \alpha_0.$$

The second step is to extend the system by adding rules for translating dynamic positive conditional formulae into positive conditional formulae and conversely. So we have two sets of rules: one for the elimination and the other for the introduction of the temporal combinators.

Elimination rules:

$$[\text{str3}] \quad \frac{\Gamma \Rightarrow \triangle(t, \pi)}{\Gamma \Rightarrow D(t)}$$

$$\frac{\Gamma \Rightarrow \triangle(t, [\, \lambda x \,.\, \phi \,])}{\Gamma \Rightarrow \phi[t/x]}$$

$$\frac{\Gamma \Rightarrow \triangle(t, \langle \lambda x \,.\, \phi \rangle)}{\Gamma, t \xrightarrow{y} z \Rightarrow \phi[y/x]} \quad y,\, z \text{ not appearing in } \Gamma,\, \phi \text{ and } t$$

$$\frac{\Gamma \Rightarrow \triangle(t, \bigcirc\, \pi)}{\Gamma, t \xrightarrow{y} z \Rightarrow \triangle(z, \pi)} \quad y,\, z \text{ not appearing in } \Gamma,\, \pi \text{ and } t$$

$$\frac{\Gamma \Rightarrow \triangle(t, \Box\, \pi)}{\Gamma,\, Trans(t, z) \Rightarrow \triangle(z, \pi)} \quad z \text{ not appearing in } \Gamma,\, \pi \text{ and } t$$

Introduction rules:

$$\frac{\Gamma \Rightarrow D(t) \quad \Gamma \Rightarrow \phi[t/x]}{\Gamma \Rightarrow \triangle(t, [\, \lambda x \,.\, \phi \,])}$$

$$\frac{\Gamma \Rightarrow D(t) \quad \Gamma, t \xrightarrow{y} z \Rightarrow \phi[y/x]}{\Gamma \Rightarrow \triangle(t, \langle \lambda x \,.\, \phi \rangle)} \quad y,\, z \text{ not appearing in } \Gamma,\, \phi \text{ and } t$$

$$\frac{\Gamma \Rightarrow D(t) \quad \Gamma, t \xrightarrow{y} z \Rightarrow \triangle(z, \pi)}{\Gamma \Rightarrow \triangle(t, \bigcirc\, \pi)} \quad y,\, z \text{ not appearing in } \Gamma,\, \pi \text{ and } t$$

$$\frac{\Gamma \Rightarrow D(t) \quad \Gamma,\, Trans(t, z) \Rightarrow \triangle(z, \pi)}{\Gamma \Rightarrow \triangle(t, \Box\, \pi)} \quad z \text{ not appearing in } \Gamma,\, \pi \text{ and } t$$

Let us call $\mathcal{DP}$ the full system and $\vdash^{\mathcal{DP}}$ the associated deduction relation.

Proposition 25 *If $dsp = (D\Sigma, AX)$ is a dynamic specification where the formulae in AX are dynamic positive conditional, then for every ϕ which is dynamic positive conditional:*

$$Mod(dsp) \models \phi \qquad \textit{iff} \qquad AX \cup TRANS \vdash^{\mathcal{DP}} \phi.$$

Proof. *It is easy to verify the soundness of $\mathcal{DP}$, therefore we omit this part. To see its completeness, let $dsp' = (D\Sigma', AX')$ be as above.*

From Lemma 23, if $Mod(dsp) \models \phi$ then $Mod(dsp') \models T(\phi)$. We have seen that $T(\phi)$ is equivalent to a finite set of positive conditional formulae that we denote by $T''(\phi)$; the same holds for the axioms in AX', thus we obtain the set $AX'' = (\cup\{T''(\phi) \mid \phi \in AX\}) \cup TRANS$. If $T''(\phi) = \{\theta_1, \ldots, \theta_n\}$ then $Mod(dsp') \models \theta_i$, $i = 1, \ldots, n$. The completeness of the system $\mathcal{P}$ ensures that $AX'' \vdash^{\mathcal{P}} \theta_i$, $i = 1, \ldots, n$. Consider the deduction trees for θ_i, $i = 1, \ldots, n$, from AX'' in $\mathcal{P}$ (where we picture the root below the leaves). From the roots we can obtain ϕ by using (downwards) the introduction rules. Similarly, from the leaves, belonging to AX'', we can obtain axioms in AX by going upwards and using the elimination rules (while leaving the axioms in $TRANS$ unchanged). Thus, $AX \cup TRANS \vdash^{\mathcal{DP}} \phi$. $\square$

5 Classical Results and Techniques in the Dynamic Case

In the last years a body of concepts, results and techniques has been developed in the field of algebraic specifications of adt's: parameterized specifications, constructs for building complex specifications in a modular way, behavioural/observational semantics, notions of "correct implementation", development processes, theoretical tools (as deductive systems and transformational rules), software tools (as parsers, type checkers and rapid prototypers), . . . : see, e.g. [53]. Most of them can be lifted up to work on our dynamic specifications and this usually can be done easily enough and in a sound way: the dynamic features are handled in a coherent and appropriate way.

Here, as sample instances, we discuss structuring constructs and implementations for dynamic specifications, following [54,13]. The related concepts and results have been successfully used in the specification of several case studies, see e.g. [44,45,10].

A sufficiently powerful language for writing dynamic specifications in a modular way can be obtained by considering three constructs only: sum, rename and hide/export, following the pattern proposed in the literature, see [53] for a survey. Here we do not consider constructs, as the ASL **reach**, for restricting the models of a specification to those generated by some constructors, even if this restriction may be useful for dynamic specifications. Moreover we do not consider constructs, such as the ASL observe, that are used to take observational equivalence into account; the experience in applying our approach to a range of test cases suggests that we can dispense with such constructs since our logic is powerful and flexible enough to express, directly and at the right level of abstraction, the wanted properties. On the other hand, notice that the dynamic features add an extra "dimension" to some of the operators, namely sum and export, as discussed below.

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

Let $LOOSE\_ADDT$ be the class of all pairs of the above form and $SPEC\_EXPR$ be the set of all language expressions (specification expressions); then the semantics of the language is given by a function

$$\mathcal{S}: SPEC\_EXPR \to LOOSE\_ADDT.$$

$SPEC\_EXPR$ and $\mathcal{S}$ are defined inductively by the rules given below. For simplicity we do not distinguish between specifications and specification expressions, using dsp, dsp', ... for both. We use the following notation: if $\xi = (D\Sigma, \mathcal{MOD})$, we write $Sig(\xi)$ and $Mod(\xi)$ for $D\Sigma$ and $\mathcal{MOD}$ respectively. Moreover, union and containment of signatures are taken componentwise (and w.r.t. the four components: SRT, OP, PR and DS).

Simple Specifications Simple (or flat) specifications are the basic building blocks.

$(D\Sigma, AX) \in SPEC\_EXPR$, for all dynamic signatures $D\Sigma$ and $AX \subseteq F_{D\Sigma}(X)$

$\mathcal{S}[\![(D\Sigma, AX)]\!] =$
$\quad (D\Sigma, \{DA \mid DA \text{ is a } D\Sigma\text{-algebra and for all } \phi \in AX: DA \models \phi\})$

In the examples we shall use the mixfix notation $D\Sigma$ **axiom** AX, for the simple specification $(D\Sigma, AX)$.

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

$dsp_1 + dsp_2 \in SPEC\_EXPR$, for all $dsp_1, dsp_2 \in SPEC\_EXPR$

$\mathcal{S} [\![dsp_1 + dsp_2]\!] =$
$(D\Sigma, \{DA \mid DA$ is a $D\Sigma$-algebra and $DA_{|D\Sigma_i} \in Mod(\mathcal{S} [\![dsp_i]\!])$ for $i = 1, 2\})$

where $D\Sigma = D\Sigma_1 \cup D\Sigma_2$ and $D\Sigma_i = Sig(\mathcal{S} [\![dsp_i]\!])$ for $i = 1, 2$.

Notice that this construct allows us to specify static and dynamic features separately (and then combine them). More precisely, let $D\Sigma_i$ be $((SRT_i, OP_i, PR_i), DS_i)$, $i = 1, 2$, and srt be a sort such that: $srt \in SRT_1$, $srt \notin DS_1$ and $srt \in DS_2$ (hence $srt \in SRT_2$). Then we may specify the static structure of the elements of sort srt in dsp_1 and the dynamic one in dsp_2; when we consider $dsp_1 + dsp_2$ we obtain the wanted specification.

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

Rename This construct is used to avoid name-clashes when putting specifications together. We shall consider bijective renamings only. The concept of signature isomorphism is needed here; the intuitive idea is very simple: a bijective renaming of sorts, operation and predicate symbols which "preserves" arities, static/dynamic distinctions, labels, ...; see Sect. 4.2 for a formal definition.

$\rho \bullet dsp \in SPEC\_EXPR$
for all $dsp \in SPEC\_EXPR$ and all signature isomorphisms ρ

$\mathcal{S} [\![\rho \bullet dsp]\!] =$
if ρ is an isomorphism from $Sig(\mathcal{S} [\![dsp]\!])$ into $D\Sigma'$ then
$(D\Sigma', \{DA' \mid DA'$ is a $D\Sigma'$-algebra and $\rho^{-1}(DA') \in Mod(\mathcal{S} [\![dsp]\!])\})$
else undefined,
where if $D\Sigma = Sig(\mathcal{S} [\![dsp]\!])$, then $DA = \rho^{-1}(DA')$ is the $D\Sigma$-algebra defined by:

- $DA_{srt} = DA'_{\rho(srt)}$ for all sorts srt of $D\Sigma$,
- $Op^{DA} = \rho(Op)^{DA'}$ for all operation symbols Op of $D\Sigma$,
- $Pr^{DA} = \rho(Pr)^{DA'}$ for all predicate symbols Pr of $D\Sigma$.

In the examples we use the mixfix notation **rename** dsp **with** ρ.

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

$dsp_{|D\Sigma} \in \mathit{SPEC\_EXPR}$
  for all $dsp \in \mathit{SPEC\_EXPR}$ and all dynamic signatures $D\Sigma$

$\mathcal{S} [\![dsp_{|D\Sigma}]\!] =$
  if $D\Sigma \subseteq \mathit{Sig}(\mathcal{S} [\![dsp]\!])$, then $(D\Sigma, \{DA_{|D\Sigma} \mid DA \in \mathit{Mod}(\mathcal{S} [\![dsp]\!])\})$
  else undefined (see Sect. 2 for the definition of $DA_{|D\Sigma}$).

In the examples we use the mixfix notation **export** $D\Sigma$ **from** dsp and also the dual construct, **hide**, defined by:

$$\textbf{hide } H \textbf{ in } dsp \quad =_{\text{def}} \quad \textbf{export } D\Sigma \textbf{ from } dsp,$$

where: H is a set of sorts, dynamic sorts, operation symbols, predicate symbols and $D\Sigma = \mathit{Sig}(\mathcal{S} [\![dsp]\!]) - H$.

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

Example 26
Here, we specify concurrent systems where several processes, specified by PROC *below, interact by exchanging messages through a shared buffer, specified by* BUF_2 *of Ex. 21. We assume that* COM *and* MEM *specify, respectively, the commands executed by the processes and their local memory (containing natural values). We assume a loose semantics.*

spec PROC =
  enrich COM + MEM **by**
  dsorts *proc*
  opns
    $\langle \_, \_ \rangle : com \times mem \to proc$
    $TAU : \to lab(proc)$
    $SEND, REC : nat \to lab(proc)$
  axioms
    $D(\langle c, m \rangle)$
    ... other axioms ...

A process is a pair whose components are a command-list and a local memory; it can perform internal actions, i.e. transitions labelled by TAU*, or visible actions: sending and receiving natural numbers.*

spec SYST =
  enrich PROC + BUF_2 **by**
  sorts *procs*
  dsorts *syst*
  opns
    $\emptyset \colon \to procs$
    $\_ \mid \_ \colon proc \times procs \to procs$
    $\_ \mid \_ \colon procs \times buf \to syst$
    $Tau \colon \to lab(proc)$
  axioms
    $D(\emptyset) \qquad D(p \mid ps) \qquad D(ps \mid b)$
    - - the order of the component processes does not matter
    $p \mid p' \mid ps = p' \mid p \mid ps$

SYST *defines the structure of our systems: in term-generated models, these are composed by a buffer and zero or more processes.* SYST_1*,* SYST_2 *and* SYST_3*, below, specify how the single components of a system contribute to its overall behaviour.*

spec SYST_1 =
  enrich SYST **by**
  axioms
    $p \xrightarrow{TAU} p' \supset p \mid ps \mid b \xrightarrow{Tau} p' \mid ps \mid b$
    $p \xrightarrow{SEND(n)} p' \;\wedge\; b \xrightarrow{I(n)} b' \supset p \mid ps \mid b \xrightarrow{Tau} p' \mid ps \mid b'$
    $p \xrightarrow{REC(n)} p' \;\wedge\; b \xrightarrow{O(n)} b' \supset p \mid ps \mid b \xrightarrow{Tau} p' \mid ps \mid b'$

In the models of SYST_1*, and within a system, we allow the buffer and any process to synchronize. Moreover any transition of a process-buffer pair produces a transition of the whole system and the same happens for a TAU-transition of a single process, i.e. interleaving is allowed. But this is not the only way in which the system can evolve: the axioms do not forbid, say, that two processes synchronize and exchange messages directly. (This is a consequence of adopting a loose semantics.)*

spec SYST_2 =
  enrich SYST **by**
  axioms
    $s \xrightarrow{l} s' \supset$
      $(\exists\, p, p', ps, b \,.\, p \xrightarrow{TAU} p' \;\wedge\; s = p \mid ps \mid b \;\wedge\; s' = p' \mid ps \mid b) \vee$
      $(\exists\, n, p, p', ps, b, b' \,.\, p \xrightarrow{SEND(n)} p' \;\wedge\; b \xrightarrow{I(n)} b' \;\wedge$
        $s = p \mid ps \mid b \;\wedge\; s' = p' \mid ps \mid b') \vee$
      $(\exists\, n, p, p', ps, b, b' \,.\, p \xrightarrow{REC(n)} p' \;\wedge\; b \xrightarrow{O(n)} b' \;\wedge$
        $s = p \mid ps \mid b \;\wedge\; s' = p' \mid ps \mid b')$

In the models of SYST_2*, if a system makes a transition this must be the result* *either of the TAU-move of a single process, or of the synchronized move of a process-buffer pair. Thus, idleness is the only alternative to interleaving (and indeed* SYST_2 *has models with systems which do nothing).*

$$\mathrm{SYST}_3 = \mathrm{SYST}_1 + \mathrm{SYST}_2$$

A model of SYST_3 *must satisfy the axioms in* SYST_1 *and those in* SYST_2*; therefore, now, a system performs a transition* iff *so does a component process (alone or together with the buffer) and interleaving is* the *rule.*

Finally, we can hide the dynamic features of the components (processes and buffer) of the systems specified by SYST_3*. Then, the resulting specification,* SYST_4 *below, describes (a class of) non-deterministic sequential systems. In Sect. 5.2, we shall see that* SYST_3 *can be regarded as an* implementation *of* SYST_4 *(corresponding to the implementation of sequential systems by groups of parallel processes). Notice that this is just a simple example showing the possibilities of our framework, it is not an example on how one should proceed in designing a complex system. Hiding is obtained as follows: $Sig(\mathrm{SYST}_3)$ is $(\Sigma_3, \{syst, buf, proc\})$; then let $D\Sigma$ be $(\Sigma, \{syst\})$, where*

$$\begin{aligned}\Sigma \; = \Sigma_3 - \; & (\{lab(proc), lab(buf), TAU, SEND, REC, I, O\} \cup \\ & \{\_ \xrightarrow{\;} \_ \textit{ of type } proc \times lab(proc) \times proc \textit{ and } buf \times lab(buf) \times buf\}).\end{aligned}$$

Then the wanted specification is:

$$\mathrm{SYST}_4 \;=\; \textbf{export } D\Sigma \textbf{ from } \mathrm{SYST}_3$$

5.2 Implementation of Dynamic Specifications

Here we follow once more the approach in [54], see also [53] and the references in [53]. Let *sp* and *sp′* be two classical (or static) specifications; when is *sp* implemented by *sp′*? There are, at least, two criteria to consider:

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

Formally, we have that *sp* is *implemented* by *sp′* with respect to α, a function from specifications into specifications, (written $sp \rightsquigarrow\!\!\rightsquigarrow\!\!\rightsquigarrow^{\alpha} sp'$) iff

$$Mod(\alpha(sp')) \subseteq Mod(sp).$$

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

Clearly not all specification functions are acceptable; for example if α adds all predicates and operations of sp to sp' we have a kind of trivial implementation. However, the definition above includes as particular cases the various definitions proposed in the literature. Usually α is a combination of the various operations for structuring specifications (see Sect. 5.1). When α is a composition of a renaming, an enrichment with derived operations and predicates, an export and an enrichment with axioms, we have the so called implementation by rename-enrich-restrict-identify of [22,23] which corresponds, within the framework of abstract data types, to Hoare's idea of implementation of concrete data types.

The above definition, when used in our setting, yields a reasonable notion of implementation for dynamic specifications, covering relevant applications, see e.g. [10] and there is no need for a notion of observability.

Definition 27 *Given two dynamic specifications dsp and dsp' and a function α, from dynamic specifications into dynamic specifications, defined by a combination of the constructs of Sect. 5.1:*

> *dsp is* implemented *by dsp' w.r.t. α (or dsp' is an* implementation *of dsp w.r.t. α), written $dsp \leadsto\!\!\leadsto^{\alpha} dsp'$, iff $Mod(\alpha(dsp')) \subseteq Mod(dsp)$.*

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

- α does not add axioms defining the transition predicates of dsp'; then we have a *static* implementation, which concerns just the static parts of the specification (for example, the states and the labels of the transitions);
- α redefines the transitions of dsp' composing them sequentially, by adding axioms like $d_1 \xrightarrow{l_1} d_2 \xrightarrow{l_2} d_3 \ldots \xrightarrow{l_{n-1}} d_n \supset d_1 \xRightarrow{l} d_n$ ($\longrightarrow$ transition predicate in dsp' and $\Longrightarrow$ transition predicate in dsp); we have an *action-refining* implementation, because the transitions of the dynamic elements of dsp are realized by sequences of transitions in dsp'.
- α does not change dynamic sorts into static ones, nor static sorts into dynamic ones: we have a *dynamics-preserving* implementation.

Example 28 Implementations of dynamic specifications
BUF$_3$ *cannot be implemented by* BUF$_1$ *for any static α, since in* BUF$_1$ *the buffers may go on forever in receiving new values, thus violating two of the requirements in* BUF$_3$*: not receiving a new copy of a value until the previous one has been returned, and eventually returning the received values. On the other hand,* BUF$_3$ *can be statically implemented by* BUF$_4$ *below (buffers which*

can contain one value at most).

spec BUF_4 =
  enrich NAT **by**
  dsorts *buf*
  opns
    $Empty\colon \to buf$
    $C\colon nat \to buf$
    $I, O\colon nat \to lab(buf)$
  axioms
    $\neg\, 0 = Succ(n) \qquad Succ(n) = Succ(m) \supset n = m$ -- as in BUF_3
    $C(n) \xrightarrow{O(n)} Empty \qquad Empty \xrightarrow{I(n)} C(n)$
    $C(n) \xrightarrow{l} b \supset (\exists\, n\, .\, l = O(n) \,\wedge\, b = Empty)$
    $Empty \xrightarrow{l} b \supset (\exists\, n\, .\, l = I(n) \,\wedge\, b = C(n))$

Consider now BUF_5 *and* BUF_6 *given by:*

spec BUF_5 =
  enrich BUF_4 **by**
  opns
    $Get\colon buf \to nat$
    $Put\colon nat \times buf \to buf$
    $Remove\colon buf \to buf$
  axioms
    $\neg\, D(Get(Empty)) \qquad Get(C(n)) = n$
    $\neg\, D(Put(n, C(m))) \qquad Put(n, Empty) = C(n)$
    $\neg\, D(Remove(Empty)) \qquad Remove(C(n)) = Empty$

spec BUF_6 = **hide** C: *nat* $\to$ *buf* **from** BUF_5.

It is easy to see that $Mod(\mathrm{BUF}_6) \subseteq Mod(\mathrm{BUF}_3)$*; thus* BUF_3 *is implemented by* BUF_4 *w.r.t.* α *which is the composition of* **hide** $C\colon nat \to buf$ **from** _ *and*
  enrich _ **by opns** $Get\colon buf \to nat \ldots$ **axiom** $\neg\, D(Get(Empty)) \ldots$.

SYST_3 *is an implementation of* SYST_4*, where* α *is* **export** $D\Sigma$ **from** _. *Here* α *does not satisfy any of the conditions listed above. This example is interesting because it shows that our definition covers cases such as: implementation of a sequential system by another composed of several processes in parallel; however, it is* not *an example of good design-methodology.*

Now we give an example of action-refining implementation: BUF_4 *is implemented by a low-level buffer* BUF_7 *which interacts with the external world by sending and receiving sequences of "units" represented by the symbol* 1*. Thus* BUF_7 *is also an implementation of* BUF_3.

spec UNITS =
  sorts *units*
  opns
    $\Lambda \colon \rightarrow units$
    $1\_ : units \rightarrow units$

spec BUF$_7$ =
  enrich UNITS **by**
  dsorts *buf*
  opns
    $R, S, C \colon units \rightarrow buf$
  - - here there are only 3 possibel "states" for a buffer:
  - - receiving/sending a sequence of units (R, S) and idling (C)
    $IU, OU \colon \rightarrow lab(buf)$
  - - input/output of one unit
    $START\_I, END\_I \colon \rightarrow lab(buf)$
  - - starting/ending the input of a sequence of units
    $START\_O, END\_O \colon \rightarrow lab(buf)$
  - - starting/ending the output of a sequence of units
  axioms

$$C(u) \xrightarrow{START\_O} S(u) \qquad S(1u) \xrightarrow{OU} S(u) \qquad S(\Lambda) \xrightarrow{END\_O} C(\Lambda)$$
$$C(\Lambda) \xrightarrow{START\_I} R(\Lambda) \qquad R(u) \xrightarrow{IU} R(1u) \qquad R(u) \xrightarrow{END\_I} C(u)$$

BUF$_7$ *is an implementation (action-refining of* BUF$_4$*); indeed consider the following specifications:* BUF$_8$ = BUF$_7$[$nat/units, 0/\Lambda$]

spec BUF$_9$ =
  enrich BUF$_8$ **by**
  opns
    $Succ \colon nat \rightarrow nat$
    $I, O \colon nat \rightarrow lab(buf)$
    $Empty \colon \rightarrow buf$
  preds
    $\_ \xrightarrow{\_} \_ \colon buf \times lab(buf) \times buf$
  axioms

$$Succ(u) = 1u \qquad Empty = C(\Lambda)$$
$$\{\ C(1^n) \overset{START\_O}{\Longrightarrow} b_1 \ \wedge\ b_1 \overset{OU}{\Longrightarrow} b_2 \ \wedge\ \ldots\ \wedge\ b_n \overset{END\_O}{\Longrightarrow} C(\Lambda) \supset$$
$$C(1^n) \xrightarrow{O(1^n)} Empty,$$
$$Empty \overset{START\_I}{\Longrightarrow} b_1 \overset{IU}{\Longrightarrow} \ldots\ b_n \overset{END\_I}{\Longrightarrow} C(1^n) \supset C(Empty) \xrightarrow{I(1^n)} C(1^n)$$
$$\mid \quad n \geq 1\ \}$$

where 1^n *stands for* $\underbrace{1 \ldots 1}_{n \ \ times} \Lambda$.

spec BUF$_{10}$ =
hide $\Longrightarrow, \Lambda, \_\cdot\_, 1\_, R, S, IU, OU, START\_I, START\_O, END\_I, END\_O$ **in** BUF$_9$

Now it is easy to see that $Mod(\mathrm{BUF}_{10}) \subseteq Mod(\mathrm{BUF}_4)$.

Once we have a definition of implementation, it is interesting to study its relations with the structure of specifications. We limit ourselves to implementations where α is the identity function, using for this the notation $dsp \sim\sim\!\!> dsp'$ (notice that dsp and dsp' must have the same signature). This is not a restriction: from the properties of $\sim\sim\!\!>$ we can derive properties about $\sim\sim\!\!>^{\alpha}$, because α is defined as a combination of the structuring operations; indeed, $dsp \sim\sim\!\!>^{\alpha} dsp'$ iff $dsp \sim\sim\!\!> \alpha(dsp')$.

It is easy to verify the following

Proposition 29

(i) $\sim\sim\!\!>$ *is a partial order (i.e. it is reflexive, antisymmetric and transitive).*
(ii) All specification operators are monotonic w.r.t. $\sim\sim\!\!>$ *: if* dsp, dsp', dsp'' *are specifications,* $D\Sigma$ *is contained in* $Sig(dsp)$, ρ *is a signature isomorphism and* $dsp \sim\sim\!\!> dsp'$, *then*
$dsp + dsp'' \sim\sim\!\!> dsp' + dsp''$;
$dsp_{|D\Sigma} \sim\sim\!\!> dsp'_{|D\Sigma}$;
$\rho \bullet dsp \sim\sim\!\!> \rho \bullet dsp'$. □

6 Concluding Remarks

We have presented an algebraic framework allowing to specify, in what we think is a convenient way, both "classical" (or "static") and "dynamic" data types. These last can model concurrent/parallel/reactive processes/systems.

In our opinion there are some strong points in our approach.

- Specifications follow the well established pattern of algebraic specifications for adt's; indeed what we have is an *extension* of the classical framework. This extension is "conservative" as classical definitions, properties and results are a particular case of ours (precisely the case when there are no dynamic sorts). It is also sound as it results in an institution.
- It has been tested in practice on non-trivial examples and it has proved successful: it is easy to write specifications and in many cases it is rather straightforward to prove properties of the specified systems, either using the deductive systems we have provided, or in the style of ordinary mathematics.

  As examples of applications of the proposed formalism: in [8] it supports a metalanguage for expressing requirement of reactive systems, in [44,45] it has been used for the specifications of two industrial case studies and recently in [10] it has been satisfactorily used to solve a specification problem proposed by Lamport and Broy as a common case study at a Dagstuhl Seminar [15].

– With respect to other approaches used in the specification of concurrent/ reactive systems, ours
  · offers *more abstract specifications* (and this is a well known benefit of the algebraic approaches);
  · allows to express, within the same formalism, both *requirement specifications* (i.e. specifications with a loose semantics) and *design specifications* (i.e. those with an initial semantics);
  · concerns the specification of *types of systems* rather than the specification of *one system*;
  · allows to *define* the kind of parallelism and process-interaction that we want.

An apparent weakness is that we do not address the issue of observational (or behavioural) semantics; indeed, here we have considered only the initial and the loose semantics. Actually, observational semantics for processes, defined on the associated labelled transition systems, may be used to define models for a dynamic specification of such processes. This is shown in [4,11] for conditional dynamic specifications: given an observational semantics for the dynamic elements (e.g. strong or weak bisimulation) one obtains a model where two terms of dynamic sort are identified iff they are observationally equivalent. On the other hand, the experience in using the SMoLCS method (see e.g. [44,45,10]) suggests that we do not need observational semantics to express requirements over processes or to reason about implementation. This is due to the power and flexibility of our logic, when used to express properties in requirement specifications.

Comparisons with other approaches

In the last years, several authors have proposed the use of modal or temporal logics for the specification of data types or have considered algebraic specifications for describing and modelling processes, concurrent systems and "objects".

In some cases, like in [36,50], the aim is exploiting the power of temporal logics to obtain specifications of classical data types which are "stronger", in the sense that the class of models is smaller; in particular, one can obtain monomorphic specifications [36]. More frequent is, however, the use of temporal or modal logics to express time-dependent properties of data types or object types. For instance, in [26] usual specifications (with positive-conditional axioms) are considered and a temporal logic (CTL, [25]) is used just to express properties of data types related to some operations which are singled out as "update operations".

However, we cannot restrict ourselves here just to the work where temporal or modal logics are used in connection with algebraic specifications, but must consider also the different approaches to using algebraic specifications for describing and modelling processes, concurrent systems and/or objects. In relating these approaches to our work we think that we can distinguish, roughly, three groups, characterized by **A1**, **A2** and **A3** below.

A1 Just the data used by the processes are given by means of algebraic specifications of adt's, while concurrency is handled in other ways (e.g. by using a CCS like language as in LOTOS [32], stream processing functions as in [14] or Petri nets as in [46]).
A2 Processes are elements of one (or some of) the sorts of algebraic structures, of some kind, that are specified axiomatically; here a specification defines either one structure or a class of structures. (Clearly our approach falls within this category.)
A3 Data types themselves are regarded as dynamically changing. This is the so-called "dynamic adt's" approach, see e.g. [12,24].

In this last approach models are sequences of algebraic structures (thus there is a similarity with the way linear 1^{st} order temporal logic is usually interpreted – see Sect. 1). The reason is that these models are primarily intended to capture entities (like Ada packages or objects) with a local state and where the meaning of (some of) the operations is state-dependent. Alternatively, the operations are regarded as parts of the elements, rather than something shared by all elements; this is a view which is typical to object-oriented languages: an object stack has its own operation *Top* as opposed to the usual situation where there is a (single) operation *Top* acting on all stacks. The point of view and the main concern are therefore different from ours; indeed objects cannot be modelled within our framework in a straightforward way (but see [43] for an operational model for an object-oriented language using a restricted form of our formalism). However, when the aim is to model concurrent systems we think that our framework has at least the advantage that it is simpler to reason on a single algebra, especially at the logical level.

For the approaches characterized by either **A1** or **A2**, one can consider an additional distinction according to whether their main concern are design or requirement specifications. A survey of these approaches, mainly for what concerns design specifications, can be found in [9], while [11] contains a detailed comparison between our formalism and other proposals. Here we discuss only some of the proposals in the literature: those that seem more significant w.r.t. the topics presented in this paper.

Fiadeiro and Maibaum, in several papers – see e.g. [28], have presented a specification formalism for processes and objects that combines temporal logics and algebraic specifications. The main differences with our proposal are: they specify systems rather than types of systems (see above); there is a separation between the algebraic and the temporal part: the first is used for the data handled by the processes and the second for the process properties (thus their approach belongs to group **A1**); parallel composition of (the specifications of) several processes is usually obtained by "external (i.e. out of the formalism) tools" such as limits of diagrams in an appropriate category of specifications, while in our approach parallel composition is just an operation defined by axioms in our logic (see the examples in Sect. 5.1).

The work by Fiadeiro and Maibaum is mainly about requirement specifications. Another example of requirement specifications within **A1** is the work of Broy in [14]. There the requirements on processes are expressed by formulae of a linear time temporal logic, whose models are functions over streams (stream processing functions).

The object specification logic of [47] is based on a variant of linear temporal logic and covers requirement specifications; at least it supports **A1**. The underlying models for single objects and for systems of objects are linear Kripke structures, where the states/situations are sets of open atomic formulas (stating that the objects – represented by free variables – are either performing some actions or have some attribute). Clearly, the choice of linear models implies that nondeterminism cannot be handled appropriately, except by confusing nondeterminism within a process with the one arising from the various ways of realizing a requirement (one cannot write a specification whose only model is a process that can perform either α or β, but only a specification which has two models: a process that can perform only α and a process that can perform only β; then, these two processes can be regarded as two different behaviours of some "higher level" process). This formalism does also contain some of the features in **A2**: one can express temporal properties of different objects in a system of objects (but cannot use equations over objects).

A formalism for design specifications fulfilling **A2** is Meseguer's Rewriting Logics, see [39]. Rewriting Logics allows to specify types of dynamic elements, whose activity is characterized by a transition relation. Here, however, transitions correspond to rewriting steps, thus they have no labels and do not represent action-capabilities but effective actions (whose occurrence cannot be conditioned by the external world). Moreover the static part (the data) is fully specified before the dynamic one. Also the semantics is different: the structure associated with a specification in Rewriting Logic, expressed in term of category theory, does not model processes through labelled transition trees, but by means of the sets of their possible behaviours (sequences of transitions, i.e. linear proofs) modulo the ordering of independent transitions. In the end we can say that specifications in Rewriting Logics correspond to a proper subclass of conditional dynamic specifications, namely those where transitions have no distinct labels and one adds axioms for the transition relation stating that it corresponds to rewriting steps (congruence axioms are an example).

Goguen's algebraic specifications with hidden sorts, see e.g. [31], is another formalism along this line. The elements of the hidden sorts are characterized by a set of operations returning elements of non-hidden (visible) sort; an equational specification determines a behavioural model where the terms of hidden sort are identified iff they cannot be distinguished by some visible context. Hidden sorts specifications may be used to represent dynamic elements as follows: the sorts of dynamic elements are hidden and the visible operations correspond to the possible interactions with the external world. This approach works in a reasonable way only for simple processes, i.e. the ones without inner concurrency; indeed [31] suggests to compose process specifications to get specifications of concurrent systems by using, as in [28], limits of diagrams in a category of specifications.

Acknowledgement

The present work is strictly connected with other researches carried out in Genova. The authors thank Egidio Astesiano for many fruitful discussions on this and related topics and Eugenio Moggi for suggestions on the deductive systems in Sect. 4. Thanks also to the referees of [19] and of the draft version of the present paper for helpful comments.

References

[1] M. Abadi. The Power of Temporal Proofs. *TCS*, 65:25–83, 1989.

[2] E. Astesiano and M. Cerioli. On the Existence of Initial Models for Partial (Higher Order) Conditional Specifications. In J. Diaz and F. Orejas, editors, *Proc. TAPSOFT'89, Vol. 1*, number 351 in Lecture Notes in Computer Science, pages 74 – 88. Springer Verlag, Berlin, 1989.

[3] E. Astesiano and M. Cerioli. Relationships between Logical Frameworks. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, number 655 in Lecture Notes in Computer Science, pages 126–143. Springer Verlag, Berlin, 1993.

[4] E. Astesiano, A. Giovini, and G. Reggio. Observational Structures and their Logic. *TCS*, 96:249–283, 1992.

[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. On the Specification of the Firing Squad Problem. In *Proc. of the Workshop on The Analysis of Concurrent Systems, Cambridge, 1983*, number 207 in Lecture Notes in Computer Science, pages 137–156. Springer Verlag, Berlin, 1985.

[7] E. Astesiano and G. Reggio. An Outline of the SMoLCS Approach. In M. Venturini Zilli, editor, *Mathematical Models for the Semantics of Parallelism, Proc. Advanced School on Mathematical Models of Parallelism, Roma, 1986*, number 280 in Lecture Notes in Computer Science, pages 81–113. Springer Verlag, Berlin, 1987.

[8] E. Astesiano and G. Reggio. A Metalanguage for the Formal Requirement Specification of Reactive Systems. In J.C.P. Woodcock and P.G. Larsen, editors, *Proc. FME'93: Industrial-Strength Formal Methods*, number 670 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1993.

[9] E. Astesiano and G. Reggio. Algebraic Specification of Concurrency (invited lecture). In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, number 655 in Lecture Notes in Computer Science, pages 1–39. Springer Verlag, Berlin, 1993.

[10] E. Astesiano and G. Reggio. A Case Study in Friendly Specifications of Concurrent Systems (Lamport & Broy's Specification Problem Presented at the Dagstuhl Seminar "Specification and Refinement of Reactive Systems – A Case Study"). Technical Report DISI–TR–94–21, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1994.

[11] E. Astesiano and G. Reggio. Algebraic Dynamic Specifications: An Outline. Technical Report DISI–TR–95–08, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1995.

[12] E. Astesiano and E. Zucca. D-oids: a Model for Dynamic Data Types. *Mathematical Structures in Computer Science*, 1995. To appear.

[13] J.A. Bergstra, J. Heering, and P. Klint. Module Algebra. Technical Report CS-R8617, Centre for Mathematics and Computer Science, Amsterdam, 1986.

[14] M. Broy. Specification and Top Down Design of Distributed Systems. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Proc. TAPSOFT'85, Vol. 1*, number 185 in Lecture Notes in Computer Science, pages 4–28. Springer Verlag, Berlin, 1985.

[15] M. Broy and L. Lamport. Specification Problem. Distributed to the partecipants of the Dagstuhl Seminar on "Specification and Refinement of Reactive Systems: A Case Study", 1993.

[16] M. Broy and M. Wirsing. Partial Abstract Types. *Acta Informatica*, 18:47–64, 1982.

[17] R.M. Burstall and J.A. Goguen. Introducing Institutions. In E. Clarke and D. Kozen, editors, *Logics of Programming Workshop*, number 164 in Lecture Notes in Computer Science, pages 221–255. Springer Verlag, Berlin, 1984.

[18] M. Cerioli and G. Reggio. Algebraic Oriented Institutions. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology (AMAST'93)*, Workshops in Computing. Springer Verlag, London, 1993.

[19] G. Costa and G. Reggio. Abstract Dynamic Data Types: a Temporal Logic Approach. In A. Tarlecki, editor, *Proc. MFCS'91*, number 520 in Lecture Notes in Computer Science, pages 103–112. Springer Verlag, Berlin, 1991.

[20] N. J. Cutland. *Computability.* Cambridge University Press, Cambridge, 1983.

[21] H. D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic.* Springer Verlag, Berlin, 1984.

[22] H.D. Ehrich. On the Realization and Implementation. In *Proc. MFCS'81*, number 118 in Lecture Notes in Computer Science, pages 271–280. Springer Verlag, Berlin, 1981.

[23] H. Ehrig, H.J. Kreowski, B. Mahr, and P. Padawitz. Algebraic Implementation of Abstract Data Types. *TCS*, 20:209–263, 1982.

[24] H. Ehrig and F. Orejas. Dynamic Abstract Data Types: An Informal Proposal. *Bulletin of the EATCS*, (53), 1994.

[25] A.E. Emerson. Temporal and Modal Logic . In J. van Leeuwen, editor, *Handbook of Theoret. Comput. Sci.*, volume B, pages 997–1072. Elsevier, 1990.

[26] Y. Feng and J. Liu. Temporal Approach to Algebraic Specifications. In *Proc. Concur'90*, number 485 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1990.

[27] S. Ferrua and G. Reggio. A Guide to the Use of the SMoLCS Methodology. Technical report, Progetto ENEL-CRA – DISI: "Tecniche formali di specifica di sistemi concorrenti basate su linguaggi algebrici", 1995.

[28] J. Fiadeiro and T. Maibaum. Temporal Theories as Modularization Units for Concurrent System Specification. *Formal Aspects of Computing*, 4(3):239–272, 1992.

[29] J.W. Garson. Quantification in Modal Logics. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logics II.* Reidel Publ. Co., 1984.

[30] J. Goguen and J. Meseguer. Models and Equality for Logic Programming. In *Proc. TAPSOFT'87, Vol. 2*, number 250 in Lecture Notes in Computer Science, pages 1–22. Springer Verlag, Berlin, 1987.

[31] J.A. Goguen and R. Diaconescu. Towards an Algebraic Semantics for the Object Paradigm. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification*, number 785 in Lecture Notes in Computer Science, pages 1–29. Springer-Verlag, Berlin, 1994.

[32] I.S.O. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS 8807, International Organization for Standardization, 1989.

[33] F. Kröger. *Temporal Logic of Programs.* EATCS Monographs in Theoret. Comp. Sci. Springer Verlag, Berlin, 1987.

[34] F. Kröger. On the Interpretability of Arithmetic in Temporal Logic. *TCS*, 73:47–60, 1990.

[35] L. Lamport. Specifying Concurrent Program Modules. *ACM TOPLAS*, (3):190 – 222, 1983.

[36] F. Lesske. Constructive Specifications of Abstract Data Types Using Temporal Logics. In *Proc. Logical Foundations of Computer Science, Tver '92*, number 620 in Lecture Notes in Computer Science, pages 269–280. Springer Verlag, Berlin, 1992.

[37] Z. Manna and A. Pnueli. The Anchored Version of the Temporal Framework. 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. Springer Verlag, Berlin, 1989.

[38] Z. Manna and A. Pnueli. *The Temporal Logics of Reactive and Concurrent Systems.* Springer Verlag, New York, 1992.

[39] J. Meseguer. Rewriting as a Unified Model of Concurrency. *TCS*, 96, 1992.

[40] R. Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.

[41] E. Moggi. *The Partial Lambda-Calculus.* PhD thesis, University of Edimburgh, 1988.

[42] V.R. Pratt. Dynamic Algebras and the Natura of the Induction. In *12th ACM Symposium on Theory of Computation, Los Angels*, 1980.

[43] G. Reggio and A. Casarino. A Semantics for TROLL*light* Using Algebraic Dynamic Specifications. Technical Report DISI-TR-95-10, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1995. Available by anonymous ftp at `ftp.disi.unige.it`, directory `pub/reggio`.

[44] G. Reggio and E. Crivelli. Specification of a Hydroelectric Power Station: Revised Tool-Checked Version. Technical Report DISI–TR–94–17, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1994.

[45] G. Reggio and V. Filippi. Specification of a High-Voltage Substation: Revised Tool-Checked Version. Technical Report DISI-TR-95-09, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1995.

[46] W. Reisig. Petri Nets and Algebraic Specifications. *TCS*, 80:1–34, 1991.

[47] A. Sernadas, C. Sernadas, and J.F. Costa. Object Specification Logic. *J. Logic Computation*, 5(5):603 – 630, 1995.

[48] C. Stirling. Comparing Linear and Branching Time Temporal Logics. In *Temporal logics of Specification*, number 398 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1989.

[49] C. Stirling. Modal and Temporal Logics. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 2*, pages 477–563. Clarendon Press, Oxford, 1992.

[50] A. Szalas. Towards the Temporal Approch to Abstract Data Types. *Fundamenta Informaticae*, XI:49–63, 1988.

[51] A. Szalas and L. Holenderski. Incompleteness of First-Order Temporal Logic with Until. *TCS*, 57:317–325, 1988.

[52] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics (vol. I)*. Number 121 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1988.

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

[54] M. Wirsing. Proofs in Structured Specifications. In F.L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification (Proc. of the International Summer School in Marktoberdorf, Germany, 1991)*, NATO-ASI Series F 94, pages 411–442. Springer Verlag, Berlin, 1993.

Contents

1 Introduction 1

2 Partial Algebras with Predicates 7

3 Dynamic Algebras 9

4 Specifications of Abstract Dynamic-Data Types 13

4.1 The Logic 13

4.2 The institution of algebraic temporal logic 17

The signature category17 The sentence functor17 The algebra functor18 The satisfaction relations18

4.3 Deductive Systems 20

4.4 (Simple) Dynamic Specifications 26

5 Classical Results and Techniques in the Dynamic Case 33

5.1 Structured specifications 34

Simple Specifications34 Sum of specifications35 Rename35 Export36

5.2 Implementation of Dynamic Specifications 38

6 Concluding Remarks 42

Acknowledgement 46

References 46