To appear in: Proc. MFCS'91, LNCS Springer.

ABSTRACT DYNAMIC DATA TYPES: A TEMPORAL LOGIC APPROACH

GERARDO COSTA - GIANNA REGGIO

DIPARTIMENTO DI MATEMATICA - UNIVERSITÀ DI GENOVA

VIA L.B.ALBERTI 4 - 16132 GENOVA (ITALY)

INTRODUCTION

Dynamic data types are (modelled by) dynamic algebras, which are a particular kind of partial algebras with predicates. These, in turn, are just the familiar algebraic structures that are needed to interpret (many sorted) 1^{st} order logic: a family of sets (the carriers) together with a set of operations and predicates on the carriers [GM]. 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 there are special ternary predicates $\longrightarrow$; d $\xrightarrow{l}$ d' means that element d can perform a transition labelled by l into element d'. If we use a dynamic algebra to model (some kind of) processes, then we may have transition predicates corresponding to send and receive actions. A dynamic algebra for lists, may have a transition predicate corresponding to the tail operation; thus: list $\xrightarrow{l}$ list' is true, for some appropriate label l, whenever list' = tail(list). Of course we are not forced to have such predicates: when modelling processes it is natural to use them (one could even say that we need them); in the case of lists 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 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 handling partial operations, and have been dealt with in the literature, see [BW, AC] for instance. The name seems appropriate, even though it has already been used to denote structures for interpreting dynamic logic.

The question is whether dynamic algebras are of any use; we think the answer is yes. Indeed, they are, in disguise, a basic tool in the SMoLCS methodology, which has been used in practice, and for large projects, with success (see eg [AR2, AFD]). SMoLCS is a specification methodology, for specifying concurrent systems, that provides a framework for handling both ordinary (static) data types and data types with dynamic features (process types). The logical language used in SMoLCS is many sorted 1^{st} order logic with equality and transition predicates. Such a language allows reasonable specifications for many properties of concurrent systems, however it becomes cumbersome when dealing with properties involving the transitive closure of the transition relations such as (some) liveness or safety properties [L]. A really significant example would take up too much space here; a simple, but still interesting, example can be cooked up using buffers (we shall use it also in the following sections). So let us consider a set B of buffers together with the operations Put, Get and Remove: Put(e, b) adds element e to buffer b; Get(b) yields the "first" element in b and leaves b unchanged; Remove(b) removes this first value from b producing a new buffer. As example of constraints on B, Get, Put and Remove we can consider:

This work has been supported by CNR-PF-Sistemi Informatici e Calcolo Parallelo (Italy), COMPASS-Esprit-BRA-W.G. n 3264 and by a grant ENEL/CRA (Milano Italy).

(i) The buffers in B follow a LIFO policy, ie: Get(Put(e, b)) = e and Remove(Put(e, b)) = b.

(ii) If b is non-empty, b' = Remove(b) and e = Get(b), then there is an elementary transition from b to b' corresponding to "output e". Using a transition predicate, we can phrase this by saying that $b \xrightarrow{O(e)} b'$ is true (here O(e) is a label meaning "output of e"; see Sec. 2 for a more precise formulation).

(iii) The buffers in B are such that they have the capability of returning any element, say e, that they receive and mantain this capability until e is actually delivered.

Condition (i) is standard and does not need comments. (iii) is a liveness constraint: once a buffer b inputs e it will evolve (through input/output transitions) in such a way that at any "state" (or moment) either e can be output or another state can be reached in which e can 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. Finally, one way of reading (ii) is: if b is nonempty then it can always output a (stored) value; thus we have an example of a simple safety property. Being a simple one it can be easily expressed in 1st order logic (with transition predicates). Of course, with the same language one can also express properties such as (iii); but the corresponding formulae are almost unreadable. This difficulty provides the principal motivation for the present work: our aim is to find a logic which is well suited for expressing in a (reasonably) natural way properties such as (iii), but also (i) and (ii).

Various temporal logics have been proposed as an adequate tool for specifying the behaviour of concurrent systems, see eg [P, Em, K]. They allow to write concise formulae corresponding to (iii), but are not well suited for properties such as (i) or even (ii). Therefore we have been brought to a language which is, more or less, a 1st order version of CTL* with explicit transition predicates (notice that with these predicates the accessibility relations used in Kripke frames are - or can be referred to - within the language). Our choices, both for syntax and semantics, have been motivated by previous experience with the problems involved in the specification of concurrent systems [AR1, AR2]. We are now experimenting with our framework (as part of a larger project aimed at extending the algebraic approach to the specification of dynamic systems) and comparing it with similar ones, such as those reported in [FL, FM, SFSE]. We hope that a better insight into the problems will either allow us to switch to a more established setting, or provide strong arguments in favour of ours. Presently we can at least say that our framework is a sound one, indeed it forms an institution [GB].

One part of the experiment consists in testing our approach against "real" problems, in an industrial environment; the other concerns the theoretical side. In the first place we are going through the well known concepts and results concerning specifications for abstract data types (see eg [W90]) and replacing 1st order logic with ours, algebras with dynamic algebras, and so on: it appears that many concepts and results can be extended to our setting and in a natural way. Here we give a first, and concise, account of what happens in connection with existence of initial models and related proof systems. In a more complete paper [CR] we consider also structured and hierarchical specifications and implementations of specifications, in the style of [W89].

Acknowledgements. The present work is strictly connected with other researches carried out in Genova. Thanks to Egidio Astesiano for many fruitful discussions on this and related topics. Thanks also to the referees for helpful comments.

1 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 (see [BW]) and from the algebras with predicates of Goguen and Meseguer (see [GM]).

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: $\{OP_{w,srt}\}_{w\in SRT^*, srt\in SRT}$ and PR is a family of sets: $\{PR_w\}_{w\in SRT^*}$.

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

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*; ie:

- if $srt \in SRT$, then A_{srt} is a set;
- if $Op: srt_1 \times \ldots \times srt_n \to srt$, then $Op^A: A_{srt_1} \times \ldots \times A_{srt_n} \to A_{srt}$ is a (partial) function;
- if $Pr: srt_1 \times \ldots \times srt_n$, 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 $(a_1, \ldots, a_n) \in Pr^A$.

Given an SRT-indexed family of sets of variables X, the *term algebra* $T_\Sigma(X)$ is the Σ-algebra defined as as usual, with the condition that $Pr^{T_\Sigma(X)} = \varnothing$ for all $Pr \in PR$. If $X_{srt} = \varnothing$ for all $srt \in SRT$, then $T_\Sigma(X)$ is simply written T_Σ and its elements are called *ground terms*.

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

In what follows we assume that sorts and arities are respected and also that our algebras have *nonempty carriers* .

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 $h = \{ h_{srt} \}_{srt\in SRT}$ where for all $srt \in SRT$ $h_{srt}: A_{srt} \to B_{srt}$ and

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

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; then 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 C, an algebra I is *initial* in C
iff $I \in C$ and for all $A \in C$ there exists a unique homomorphism $h: I \to A$. The following holds:
if I is initial in 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 $A \in C$: $A \models t_1 = t_2$; thus $I \models D(t_1)$ iff for all $A \in C$: $A \models D(t_1)$; therefore, in general, the term algebra T_Σ is not initial in the class of all Σ-algebras;
- $I \models Pr(t_1, \ldots, t_n)$ iff for all $A \in C$: $A \models Pr(t_1, \ldots, t_n)$.

2 DYNAMIC ALGEBRAS

A *dynamic signature* $D\Sigma$ is a couple (Σ, STATE) where:

– $\Sigma = (\mathrm{SRT}, \mathrm{OP}, \mathrm{PR})$ is a predicate signature,

– $\mathrm{STATE} \subseteq \mathrm{SRT}$ (the elements in STATE are the *dynamic sorts*, ie the sorts of dynamic elements),

– for all $st \in \mathrm{STATE}$
  there exist a sort $lab(st) \in \mathrm{SRT} - \mathrm{STATE}$ such that $lab(st') = lab(st'')$ iff $st' = st''$
  and a predicate $\_ \xrightarrow{\_} \_ : st \times lab(st) \times st \in \mathrm{PR}$.

A (*dynamic*) $D\Sigma$-algebra is just a Σ-algebra; the term algebra $T_{D\Sigma}(X)$ is just $T_\Sigma(X)$.

Notation: 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, $\_ \xrightarrow{\_} \_ : st \times lab(st) \times st \in \mathrm{PR}$ means that we shall write $t \xrightarrow{t'} t''$ instead of $\longrightarrow(t, t', t'')$; ie terms of appropriate sorts replace underscores.

If DA is a $D\Sigma$-algebra and $st \in \mathrm{STATE}$, then the elements of sort st, the elements of sort lab(st) and the interpretation of the predicate $\_ \xrightarrow{\_} \_$ are respectively the states, the labels and the transitions of a labelled transition system, describing the activity of the dynamic elements of sort st. The whole activity of the dynamic elements is represented by the *maximal* labelled paths, such as

$s_0 \xrightarrow{l_0}{}^{DA} s_1 \xrightarrow{l_1}{}^{DA} s_2 \dots$ (either finite, and non-extendable, or infinite).

We denote by PATH(DA, st) the set of such paths for the dynamic elements of sort st.
If σ is the path above, then: $S(\sigma)$ denotes s_0, $L(\sigma)$ denotes l_0, $\sigma|_n$ denotes the subpath from s_n onwards (if it exists).

In what follows $D\Sigma$ will denote a generic dynamic signature (Σ, STATE), where $\Sigma = (\mathrm{SRT}, \mathrm{OP}, \mathrm{PR})$; moreover we often write: **sorts** S **dsorts** STATE **opns** OP **preds** PR for the dynamic signature (Σ, STATE), where Σ is:

$(S \cup \mathrm{STATE} \cup \{ lab(st) \mid st \in \mathrm{STATE} \}, \mathrm{OP}, \mathrm{PR} \cup \{\_ \xrightarrow{\_} \_ : st \times lab(st) \times st \mid st \in \mathrm{STATE}\})$.

Example: buffers containing natural values organized in a LIFO way

Consider the dynamic signature BUFΣ

sorts nat
dsorts buf
opns 0: $\to$ nat
Succ: nat $\to$ nat
Empty: $\to$ buf
Put: nat $\times$ buf $\to$ buf
Get: buf $\to$ nat
Remove: buf $\to$ buf
I, O: nat $\to$ lab(buf)

The elements built by the two operations I and O label the transitions corresponding to the actions of receiving and returning a value, respectively.

The buffers are modelled by the BUFΣ- algebra STACKBUF, where:

– $\mathrm{STACKBUF}_{nat} = \mathbb{N}$; $\mathrm{STACKBUF}_{buf}$ and the interpretation of the operations Empty, Put, Get and Remove are respectively the set of stacks of natural numbers and the usual operations EmptyStack, Push, Top and Pop.

– If we assume that the buffers are bounded and can contain k elements at most, then the interpretation of $\longrightarrow$ in STACKBUF is the relation consisting of the following triples (here and below the interpretation of a [predicate / operation] symbol Symb in STACKBUF, $\text{Symb}^{\text{STACKBUF}}$, is simply denoted by Symb):

$b \xrightarrow{I(n)} \text{Put}(n, b)$ for all n and all b having k-1 elements at most,

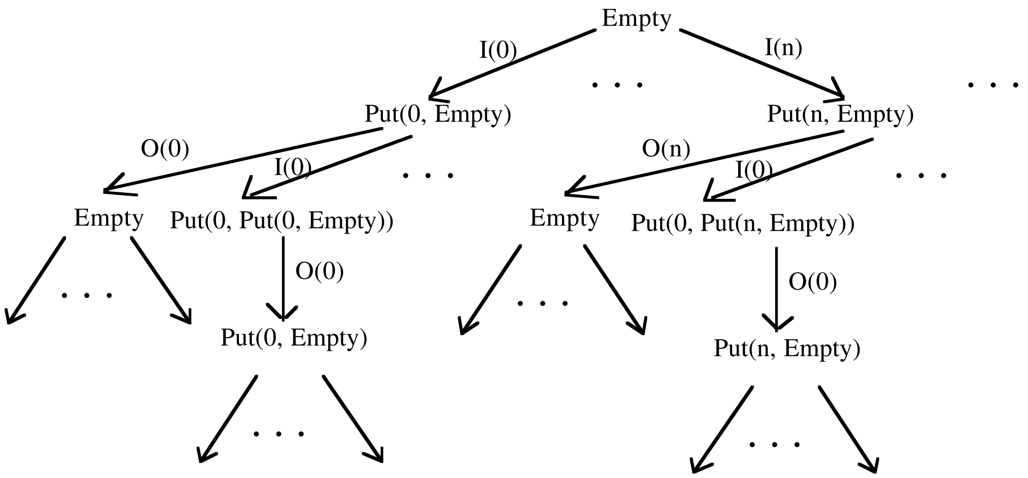
$b \xrightarrow{O(\text{Get}(b))} \text{Remove}(b)$ for all b s.t. Get(b) is defined.

– If we assume that buffers are unbounded, then $\longrightarrow$ consists of the triples:

$b \xrightarrow{I(n)} \text{Put}(n, b)$ for all n and all b,

$b \xrightarrow{O(\text{Get}(b))} \text{Remove}(b)$ for all b s.t. Get(b) is defined.

The activity of a bounded buffer, with k = 2, which is initially empty (represented by the term Empty) is given by the set of paths starting from the root of the following tree; notice that they are exactly the elements of PATH(STACKBUF, buf) with initial element the empty buffer.



End of Example

Let DA and DA' be DΣ-algebras; a *(dynamic) homomorphism* $h: DA \to DA'$ is just a homomorphism from DA into DA' as Σ-algebras. It is easy to see that, for each signature DΣ, DΣ-algebras and dynamic homomorphisms form a category, that we denote by $\text{DAlg}_{D\Sigma}$.

Homomorphisms between dynamic algebras preserve the activity of the dynamic elements; formally:

if h is as above, for all $s, l, s' \in DA$: if $s \xrightarrow{l}^{DA} s'$, then $h(s) \xrightarrow{h(l)}^{DA'} h(s')$.

If DI is initial in a class $\mathcal{D}$ of DΣ-algebras then its element have the minimum amount of activity:

$DI \models t \xrightarrow{t'} t''$ iff for all $DA \in \mathcal{D}$: $DA \models t \xrightarrow{t'} t''$.

3 A Logic for Specifying Dynamic Data Types

Following a widely accepted idea (see eg [W90]) a (static) *abstract data type* (shortly ADT) is an isomorphism class of Σ-algebras and it is usually given by a *specification*, ie a couple sp = (Σ, AX), where Σ is a signature and AX a set of 1^{st} 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:

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

In the *initial algebra approach* sp defines the ADT consisting of the (isomorphism class of the) initial elements of the class Mod(sp). 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 ADDT) is an isomorphism class of DΣ-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.

Recall that $D\Sigma = (\Sigma, STATE)$ and $\Sigma = (SRT, OP, PR)$; moreover let X be a fixed SRT-sorted family of variables s.t. for each sort srt X_{srt} is a denumerable set.

The sets of *dynamic formulae* and of *path formulae* of sort $st \in STATE$ on $D\Sigma$ and X, denoted respectively by $F_{D\Sigma}(X)$ and $P_{D\Sigma}(X, st)$, are inductively defined as follows (where $t_1, \ldots, t_n$ denote terms of appropriate sort and we assume that sorts are respected):

dynamic formulae

– $Pr(t_1, \ldots, t_n)$, $t_1 = t_2 \in F_{D\Sigma}(X)$ if $Pr \in PR$

– $\neg \phi$, $\phi_1 \supset \phi_2$, $\forall x . \phi \in F_{D\Sigma}(X)$ if $\phi, \phi_1, \phi_2 \in F_{D\Sigma}(X)$, $x \in X$

– $\Delta(t, \pi) \in F_{D\Sigma}(X)$ if $t \in T_{D\Sigma}(X)_{st}$, $\pi \in P_{D\Sigma}(X, st)$

path formulae

– $[\lambda x . \phi]$, $\langle \lambda y . \phi \rangle \in P_{D\Sigma}(X, st)$ if $x \in X_{st}$, $y \in X_{lab(st)}$, $\phi \in F_{D\Sigma}(X)$

– $\neg \pi$, $\pi_1 \supset \pi_2$, $\forall x . \pi$, $\pi_1 \mathbf{U} \pi_2 \in P_{D\Sigma}(X, st)$ if $\pi, \pi_1, \pi_2 \in P_{D\Sigma}(X, st)$, $x \in X$.

Let DA be a $D\Sigma$-dynamic algebra and $V: X \to DA$ be a variable evaluation (ie 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 $\pi \in P_{D\Sigma}(X, st)$ *holds on a path* $\sigma \in PATH(DA, st)$ *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, for a path σ, $S(\sigma)$ and $L(\sigma)$ have been defined in Sec. 2.

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, $DA, V[v/x] \models \phi$;

– $DA, V \models \Delta(t, \pi)$ iff for all $\sigma \in PATH(DA, st)$, with st 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, $DA, \sigma, V[v/x] \models \pi$;

– $DA, \sigma, V \models \pi_1 \mathbf{U} \pi_2$ iff there exists $j > 0$ s.t. $\sigma|_j$ is defined and $DA, \sigma|_j, V \models \pi_2$ 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. Validity is preserved under isomorphisms.

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

The formula $\Delta(t, \pi)$ 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 Δ and ∇ below from [S]. We anchor those formulae to states because we do not refer to a single transition system but to a whole set of them.

The formula $[\lambda x . \phi]$ holds on a path σ whenever ϕ holds of the first state of σ; similarly the formula $< \lambda x . \phi>$ holds on σ whenever ϕ holds of the first label of σ. The need for both state and edge formulae has been already discussed in [L]. Finally, $\mathbf{U}$ is the so called strong until operator. **End remarks**

In the above definitions we have used a minimal set of combinators; in practice, however, it is convenient to use other, derived, combinators; the most common are:

true, **false**, $\vee$, $\wedge$, $\exists$, defined in the usual way; $\nabla(t, \pi) =_{def} \neg \Delta(t, \neg \pi)$;

$\Diamond \pi =_{def}$ **true** $\mathbf{U}$ π (eventually π); $\Box \pi =_{def} \neg \Diamond \neg \pi$ (always π);

π_1 $\mathbf{WU}$ $\pi_2 =_{def} \pi_1$ $\mathbf{U}$ $\pi_2 \vee \Box \pi_1$ (π_1 weak until π_2); $\mathbf{m}\, \pi =_{def}$ **false** $\mathbf{WU}$ π (next π).

A few examples should clarify the meaning of the non-standard constructs in our language; in particular, example 3) should explain the role of the binders λx. We assume that: Cs is a constant symbol of state-sort st; Ps and Pl are unary predicate symbols of sort st and lab(st), respectively; x, x' and y are variables of sort st and lab(st) respectively. Moreover, for simplicity, we do not distinguish between the symbols Cs, Ps, Pl, … and their interpretations.

1) $\Delta(Cs, \Diamond < \lambda y.Pl(y)>)$ can be read: on each path from the state Cs there exists a label satisfying Pl;

2) $\nabla(Cs, \Box \Diamond [\lambda x . Ps(x)])$ can be read: there exists a path from the state Cs that has infinitely many states satisfying Ps;

3) $\Delta(Cs, \Box [\lambda x. \nabla(x, \Diamond [\lambda x'. Ps(x')])])$ can be read: 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 x' satisfying Ps.

Our framework corresponds to an institution [GB]; here we just outline the basic definitions; full details will appear in [CR]. ***dyn*** = (**DSign**, DSen, DAlg, $\models$) is an institution, where:

DSign is the category whose objects are dynamic signatures and whose morphisms are the subclass of the morphisms of predicate signatures respecting the dynamic features (ie: dynamic sorts are mapped into dynamic sorts, special sorts and predicates are mapped into the corresponding special sorts and predicates); DSen is the sentence functor: $DSen(D\Sigma)$ is the set of formulae in $F_{D\Sigma}(X)$; DAlg is the algebra functor: $DAlg(D\Sigma)$ is the category $DAlg_{D\Sigma}$; $\models$ is our validity relation.

4 DYNAMIC SPECIFICATIONS

A *dynamic specification* is a couple $sp = (D\Sigma, AX)$, where $AX \subseteq F_{D\Sigma}(X)$. The loose semantics for sp is the class of all isomorphism classes in Mod(sp); its initial semantics is the isomorphism class of the initial elements of Mod(sp).

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

Examples: we use the signature $BUF\Sigma$ defined in Sec. 2; in ex. 1 we refer to the initial semantics and in ex. 2, 3 to the loose one.

Example 1: unbounded buffers with a LIFO policy
BUF = ($BUF\Sigma$, BUF-AX1), where BUF-AX1 consists of the following axioms:

-- properties of the data contained into the buffers (the terms 0 and Succ(n) are always defined):
D(0) D(Succ(n))
-- static properties (LIFO organization of the buffers)
D(Put(n, b)) $\neg$ D(Get(Empty)) $\neg$ D(Remove(Empty))
Get(Put(n, b)) = n 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.

This specification admits initial models (see Propostion below): algebras where the carrier of sort buf is the set of unbounded stacks of natural numbers.

Example 2: a very abstract specification of buffers containing natural values; we only require the essential properties.
BUF2 = (BUFΣ, BUF-AX2) where BUF-AX2 consists of the following axioms.

-- properties of the data contained into the buffers (natural numbers):
D(0) D(Succ(n)) $\neg$ 0 = Succ(n) 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)) $\neg$ 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 the operations Get, Put and Remove are not defined as in BUF1: we only specify some of their properties; clearly such a specification is oriented to a loose semantics. If A is an algebra which is a model of BUF2, the set A_{buf} may contain, for instance, unbounded and bounded buffers, FIFO and LIFO buffers.

Example 3: a very abstract specification of buffers containing natural values where the received values are always returned.
BUF3 is given by adding the following axioms to BUF2.

b $\xrightarrow{I(n)}$ b' $\supset$ Δ(b', ($\neg$ <λ x. x = I(n)>) **WU** <λ x. x = O(n)>)
(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)

b $\xrightarrow{I(n)}$ b' $\supset$ Δ(b', ◇ <λ x. x = O(n)>)
(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 a 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. This 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 (*) and that such capability remains until the value is actually returned (**).

(*) b $\xrightarrow{I(n)}$ b' $\supset$ Δ(b', ◇ [λ b" . Out_Cap(b", n)])

(**) Out_Cap(b, n) $\supset$ Δ(b, [λ b' . Out_Cap(b', n)] **WU** <λ x. x = O(n)>)

where Out_Cap(x, y) stands for ∃ z . x $\xrightarrow{O(y)}$ z.

BUF3 specifies a subclass of the buffers defined by BUF2: the buffers where received values will, eventually, be 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 intial models of BUF1 are not included, since there each buffer admits an infinite path composed of input (push) actions only. *End of examples*

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. One can show that the same happens with formulae including existential temporal operators; for instance: $\nabla(t, \pi)$ or $\Delta(t, \Diamond\, \pi)$. However, as in the case of classical specifications, we can guarantee the existence of initial models by restricting the form of the axioms.

A formula $\phi \in F_{D\Sigma(X)}$ is *dynamic positive conditional* iff it has the form $\wedge_{i\,=\,1,\,\ldots,\,n}\, \alpha_i \supset \psi$, where: $n \geq 0$, α_i is an atom (ie of one of the forms: $t_1 = t_2$; $Pr(t_1, \ldots, t_n)$) and ψ is either an atom or has the form $\Delta(t, \pi)$ with π built using $[\ldots]$, $<\ldots>$, $\Box$, $\mathbf{m}$ only, and the formulae inside $[\ldots]$ and $<\ldots>$ are themselves dynamic positive conditional. The properties that can be specified using axioms of this kind include "usual" static properties and *safety properties.*

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

Under the hypotheses of the above proposition, we have also a deductive system which is sound and complete with respect Mod(dsp). The first step is to consider a deductive system for equational logic with partially defined terms (but recall that algebras have nonempty carriers). This can be obtained, as in [C], by considering a system which is sound and complete for the total case and modifying it as follows: suppress reflexivity of equality; allow substitution of t for x only when t is defined (rule SUB below); add rules to assert that operations and predicates are strict (rules STR below).

One such system is given by the following rules:

$$\frac{t = t'}{t' = t} \qquad \frac{t = t' \quad t' = t''}{t = t''} \qquad (\text{SUB})\ \frac{\phi[x] \quad D(t)}{\phi[t]}$$

$$\frac{D(Op(t_1,\ldots,t_n)) \qquad t_i = t_i' \ \ (i=1,\ldots,n)}{Op(t_1,\ldots,t_n) = Op(t_1',\ldots,t_n')} \qquad \frac{Pr(t_1,\ldots,t_n) \qquad t_i = t_i' \ \ (i=1,\ldots,n)}{Pr(t_1',\ldots,t_n')}$$

$$\frac{(\wedge_{i=1,\ldots,n}\, \phi_i) \supset \phi \qquad \phi_i \ \ (i=1,\ldots,n)}{\phi} \qquad (\text{STR})\ \frac{D(Op(t_1, \ldots, t_n))}{D(t_i)} \qquad \frac{Pr(t_1, \ldots, t_n)}{D(t_i)}$$

The second step is to extend the system by adding rules for the temporal operators (in the context of dynamic positive conditional formulae). Let us consider for each dynamic sort, st, of $D\Sigma$ a (new) predicate symbol $Trans_{st}\colon st \times st$ and the set of axioms $TRANS = \cup\, \{\, TRANS_{st} \mid st \in STATE \,\}$, where, if x, z, w are variables of sort st and y is a variable of sort lab(st):

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

Then, we consider the 4 rules below plus the 4 rules obtained by reversing them (ie exchanging premise with consequence):

$$\frac{\Delta(t, [\lambda\, x \,.\, \phi])}{\phi[t/x]} \qquad \frac{\Delta(t, <\lambda\, x \,.\, \phi>)}{t \xrightarrow{\ y\ } z \supset \phi[y/x]}$$

$$\frac{\Delta(t, \Box\, \pi)}{Trans(t, z) \supset \Delta(z, \pi)} \qquad \frac{\Delta(t, \mathbf{m}\, \pi)}{t \xrightarrow{\ y\ } z \supset \Delta(z, \pi)}$$

Proposition. Let $dsp = (D\Sigma, AX)$ be a dynamic specification and $\vdash$ the deduction relation associated with the full system. If the formulae in AX are dynamic positive conditional, then: $Mod(dsp) \models \phi$ iff $AX \cup TRANS \vdash \phi$. □

REFERENCES ("LNCS x" stands for Lectures Notes in Computer Science vol. x)

[AC] Astesiano E.- Cerioli M. "On the existence of initial models for partial (higher order) conditional specifications", TAPSOFT'89, LNCS 351, 1989.

[AFD] Astesiano E. - Bendix Nielsen C. - Botta N. - Fantechi A. - Giovini A. - Inverardi P. - Karlsen E. - Mazzanti F. - Storbank Pedersen J. - Zucca E. "The Draft Formal Definition of Ada" Deliverable of the CEC MAP project: The Draft Formal Definition of ANSI/STD 1815A Ada, 1987.

[AR1] Astesiano E. - Reggio G. "On the Specification of the Firing Squad Problem" Workshop on The Analysis of Concurrent Systems, Cambridge, 1983, LNCS 207, 1985.

[AR2] Astesiano E. - Reggio G. "An outline of the SMoLCS methodology" Advanced School on Mathematical Models of Parallelism 1987, LNCS 280, 1987.

[BW] Broy M. - Wirsing M. "Partial abstract data types" Acta Informatica **18** (1982).

[C] Cerioli M. "A sound and equationally-complete deduction system for partial conditional (higher order) types" 3rd Italian Conf. on Theoret. Comp. Sci. Mantova 1989, World Scientific Pub., 1989.

[CR] Costa G.- Reggio G. "Specification and implementation of abstract dynamic data types: a temporal logic approach" Technical Report, University of Genova 1991.

[Em] Emerson A.E. "Temporal and modal logic" in Handbook of Theoret. Comp. Sci. Elsevier 1990.

[FL] Feng Y. - Liu J. "Temporal approach to algebraic specifications" Concur 90, LNCS 485, 1990.

[FM] Fiadeiro J. - Maibaum T. "Describing, Structuring and Implementing Objects", Draft, presented at the REX School/Workshop on Foundations of Object-Oriented Languages, May, 1990.

[GB] Goguen J.A. - Burstall R. "Institutions: abstract model theory for specification and programming" LFCS Report Jan.'90, Dept. of Comp. Sci. Univ. of Edinburgh.

[GM] Goguen J.A. - Meseguer J. "Models and equality for logic programming" TAPSOFT'87, LNCS 250, 1987.

[K] Kröger F. "Temporal logic of programs" EATCS Monographs, Springer 1987.

[L] Lamport L. "Specifying concurrent program modules" ACM TOPLAS **5** (1983).

[P] Pnueli A. "Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends" in Current Trends in Concurrency, LNCS 224 (1986).

[S] Stirling C. "Comparing linear and branching time temporal logics" in Temporal Logic of Specification, LNCS 398 (1989).

[SFSH] Sernadas A. - Fiadeiro J. - Sernadas C. - Ehrich H.D. "Abstract object types: a temporal perspective" in Temporal Logic of Specification, LNCS 398 (1989).

[W89] Wirsing M. "Proofs in structured specifications" Preprint 1989.

[W90] Wirsing M. "Algebraic specifications" in Handbook of Theoret. Comput. Sci. vol.B, Elsevier 1990.