Dynamic Systems with Implicit State

Marie-Claude Gaudel[1], Carole Khoury[1], and Alexandre Zamulin[2]

[1] L.R.I., URA CNRS 410
Université de Paris-Sud et CNRS, Bât. 490
91405 Orsay-cedex, France
Fax 33 1 69 15 65 86
{mcg,khoury}@lri.fr
[2] Institute of Informatics Systems
Siberian Division of Russian Academy of Sciences
Novosibirsk 630090
zam@iis.nsk.su

Abstract. This paper presents a formalism of algebraic specifications with implicit state based on the concept of dynamic system. This is a synthesis of two approaches: algebraic specifications with implicit state, and abstract typed machines, developed previously by the authors. Our proposal aims at combining the advantages of these works, with a strong motivation to keep the specifications as abstract and non algorithmic as possible. In this approach a dynamic system is defined as some algebras representing the system's state, a set of access functions permitting to observe the state and a set of modifiers permitting to change the state. This formalism makes it possible to describe behaviors of systems where an internal memory evolves, without deciding at the specification level what will be stored or computed by the implementation, and without providing an algorithmic description of global changes.

1 Introduction

This paper presents a formalism of algebraic specifications with implicit state based on the concept of dynamic system. The purpose is to make easier the specification of systems with changing internal states. Such systems are not conveniently specified by classical algebraic specifications.

The version of the formalism presented here is a synthesis of two approaches:

- The one described in [18], which is an extension of the former proposals [2] and [3], motivated by the experience in writing a complex specification ([8]);
- and the one presented in [25] which is an extension of the former proposal [24], [23] attempting to combine algebraic specifications with evolving algebras [14].

The main feature of the approach presented is the definition of a dynamic system as some algebras, representing the system's states, with some access functions permitting to observe this state, and a set of modifiers permitting

J.-P. Finance (Ed.): FASE'99, LNCS 1577, pp. 114–128, 1999.

to change the state in a predetermined way. The second important feature of the approach is the definition of a formal specification method for such systems which is just a layer above classical algebraic specifications.

2 Dynamic Systems

The formalism being defined is based on the concept of implicit state à la Z ([22]) or VDM ([17]) and algebraic specifications. It is a convergence of two previous approaches known under the names AS-IS (as Algebraic Specifications with Implicit State) [18] and Typed Gurevich Machines [25]. The formalism serves for the specification of *dynamic systems* possessing a state and a number of operations for accessing and updating the state.

The signature of a system defined by AS-IS includes a part Σ which corresponds to some data types which are used for the specification of system's states and the description of possible state updates.

The system's states are defined by *elementary access functions*. The names and profiles of these functions, Σ_{eac}, are introduced in the second part of the system's signature which uses the sorts of Σ. An elementary access function is an operation, with or without arguments, which may be different in different states.

For instance in Figure 1, *counter* and *max* are elementary access functions which yield some information on the state of the system CLOCK. NAT is a used data type.

Definition A state is a Σ'-algebra where $\Sigma' = \Sigma \cup \Sigma_{eac}$.

Moreover, *dependent access functions* can be defined using the elementary access functions and the operations in Σ. The values produced by these functions depend both on the system's state and on the values of their arguments, if any. The names and profiles of these functions, Σ_{ac}, are introduced in the third part of the system's signature with the use of sorts of Σ.

In Figure 1 *delay* is a dependent access function.

A state update modifies the elementary access functions. Possible state updates are specified by *modifiers* defined in the fourth part of the system's signature, Σ_{mod}.

An update is the invocation of a modifier. It transforms a Σ'-algebra into another Σ'-algebra. An update can change the variable part of the state of a system, namely the access functions, but it must leave unchanged the data types. For this reason, we divide the class of possible states (Σ'-algebras) into subclasses called $state_A(\Sigma, \Sigma_{eac})$ which share the same (static) Σ-algebra A. Such a subclass is called the *carrier of a dynamic system*.

Definition

$\forall A \in Alg(\Sigma)$, $state_A(\Sigma, \Sigma_{eac}) = \{A' \in Alg(\Sigma \cup \Sigma_{eac}) \mid A'|_\Sigma = A\}$

Example :

System CLOCK
use NAT ** $< \Sigma, Ax >$
** *specification of the elementary access functions*
elementary accesses ** Σ_{eac}
$counter :\rightarrow Nat$
$max :\rightarrow Nat$
** *In the initial state, counter is set to 0 and max is set to 100*
Init
$counter = 0$
$max = 100$
** *specification of the dependent access functions*
accesses ** Σ_{ac}
$delay :\rightarrow Nat$
accesses axioms ** Ax_{ac}
$delay = max - counter$
** *specification of defined modifiers*
modifiers ** Σ_{mod}
$RAZ : Nat$
$Increment :$
modifiers definitions ** Def_{mod}
$RAZ(x) = counter := 0$ **and** $max := x$
$Increment =$
begin
$delay > 0$ **then** $counter := counter + 1|$
$delay = 0$ **then** $RAZ(max)$
end
end system

Fig. 1. Example of a Specification of a Dynamic System

Definition

A dynamic system, $D(A)$, of signature $< \Sigma, \Sigma_{eac}, \Sigma_{ac}, \Sigma_{mod} >$, where A is a Σ-algebra, is a 3-uple with:
- some carrier $|D(A)| = state_A(\Sigma, \Sigma_{eac})$,
- some set of dependent access functions with names and profiles defined in Σ_{ac},
- some set of defined modifiers with names and profiles defined in Σ_{mod}.

A dependent access function name $ac : s_1, ..., s_n \rightarrow s$ is interpreted in a dynamic system $D(A)$ by a map $ac^{D(A)}$ associating with each $D(A)$-algebra A' (i.e., an algebra belonging to the carrier of $D(A)$) a function $ac^{D(A)}(A') : A'_{s1} \times ... \times A'_{sn} \rightarrow A'_s$.

The operation associated with a defined modifier of Σ_{mod} is a transformation of a $D(A)$-algebra into another $D(A)$-algebra. In the example of the clock, possible updates are the updates of the value of the counter and of the max bound. It is clear that an update of these entities should not cause any change of the data type NAT.

3 Specification of a Dynamic System

Let $DS =< (\Sigma, Ax), (\Sigma_{eac}, Ax_{Init}), (\Sigma_{ac}, Ax_{ac}, \Sigma_{mod}, Def_{mod}) >$ be a dynamic system specification. It has three levels:

- The first level is a classical algebraic specification $< \Sigma, Ax >$ (cf. [5], [7]) which defines the data types used in the system. Semantics of this specification is given by the specification language used.
  The approach is relatively independent of a particular specification language. It is only required that the semantics of a specification is a class of algebras.
- The second level defines those aspects of the system's state which are likely to change, and the initial states. It includes:
  1. A signature, Σ_{eac}, which does not introduce new sorts. It defines the names and profiles of *elementary access functions*. A model of the $< \Sigma \cup \Sigma_{eac}, Ax >$ specification is a state. In the sequel we note $\Sigma' = \Sigma \cup \Sigma_{eac}$.
  2. A set of axioms, Ax_{Init}, characterizing the admissible initial states, i. e. stating the initial properties of the system.
- The third level defines some other, dependent, access functions, and the possible evolutions of the system's states in two parts :
  1. A specification of the *dependent access functions* $< \Sigma_{ac}, Ax_{ac} >$. It does not introduce new sorts and uses the elementary access functions and the operations of Σ. The specification $< \Sigma_{ac}, Ax_{ac} >$ must be hierarchically consistent with respect to $< \Sigma', Ax >$ and sufficiently complete. This last point reflects the fact that a state is completely defined (characterized) by its elementary access functions.
     A $D(A)$-algebra A' can be extended into an algebra A'', called its *extended state*, of signature $\Sigma'' = \Sigma' \cup \Sigma_{ac}$ satisfying Ax_{ac}. We denote by $Ext_{\Sigma''}(A')$ the extended state corresponding to the state A'. Any ground term of $T_{\Sigma''}$ corresponds to a value of A' since the specification of A'' does not introduce new sorts and is sufficiently complete with respect to the specification of A'. Thus, in the sequel, we use the notion of the value of a ground Σ''-term in a $D(A)$-algebra A'.
  2. A definition of the *defined modifiers*, $< \Sigma_{mod}, Def_{mod} >$. With each elementary access function ac, an *elementary modifier* "$ac :=$" is associated. Defined modifiers are defined as compositions of these elementary modifiers. The form of this definition is presented in Section 4.1.
     As sketched in the previous part, a modifier name $mod : s_1, ..., s_n$ from Σ_{mod} is interpreted in a dynamic system $D(A)$ by a map $mod^{D(A)}$ associating a $D(A)$-algebra B with each pair $< A', < v_1, ..., v_n >>$, where A' is a $D(A)$-algebra and v_i is an element of A'_{s_i}; this map must satisfy the definition of mod in Def_{mod}. We write $mod^{D(A)}(< A', < v_1, ..., v_n >>)$ for the application of $mod^{D(A)}$ to $< A', < v_1, ..., v_n >>$.

Note. To guarantee some encapsulation of the system, elementary modifiers are only usable for the definition of defined modifiers.

4 Update Expressions

4.1 Defined Modifiers

These modifiers specify the possible changes of states. In a system specification, the definition of a modifier in Def_{mod} is given in the following way:

$mod(x_1, ..., x_n) = Em$

In this definition, *mod* is the name of the modifier being defined, $x_1, ..., x_n$ are parameters, and Em is an update expression using $x_1, ..., x_n$.

For instance, in Figure 1 we have an unconditional definition of the modifier "RAZ" where $counter := 0$ is an elementary modifier which sets *counter* to zero, and a conditional definition of the modifier "Increment" which increments the counter if $delay > 0$ and sets it to zero if $delay = 0$.

The invocation of a defined modifier corresponds to an atomic change of the system's state and must be done with constant arguments:

$mod(t_i, ..., t_n)$ where $t_i \in T_{\Sigma''}$ (ground terms constructed on Σ'').

However, when using a defined modifier (for example, M1) in the definition of another modifier (for example, M2) some variables may occur:

$M2(x_1, \ldots, x_n) =$ **begin** $\ldots M1(y_1 \ldots y_m) \ldots$ **end**

Here, the x_i are the parameters of $M2$ and the terms y_j are either ground terms or terms with variables belonging to the set $\{x_1, \ldots, x_n\}$. It is the same when using an elementary modifier in the definition of another modifier.

Update expressions used for the definition of modifiers are constructed using other defined modifiers, elementary modifiers, conditional elementary modifiers, different forms of update expression composition, and the inoperant modifier **nil** which lets the state unchanged. A precise syntax is given in [9].

4.2 Elementary Modifiers

As said above, an elementary modifier "$ac :=''$ is associated with each elementary access function *ac*. If *ac* has the profile $s_1, ..., s_n \rightarrow s$, then "$ac :=''$ has the profile $s_1, ..., s_n, s$ and can be used to construct update expressions of the form $ac(exp_1, ..., exp_n) := exp$ where the exp_i are terms of sort s_i and exp is a term of sort s.

To provide a possibility of global updates of access functions, one can use variables in the exp_i. In this case, they play a role similar to that of patterns in functional programming.

The modifier "$ac :=''$ is used for the definition of a change of state in the following way:

$\forall y_1, \ldots, y_p\ [ac(\pi_1, \ldots, \pi_n) := R]$

where the variables of π_i, for $i \in [1..n]$, and those of R belong to $\{y_1, \ldots, y_n, x_1, \ldots, x_q\}$, where $(x_1, \ldots, x_q)$ are the parameters of the modifier being defined.

It is possible to have no quantification. In this case, π_i and R are ground terms. Then the expression

$ac(\pi_1, ..., \pi_n) := R$

indicates that ac should be updated at the point $< \pi_1, ..., \pi_n >$ by assigning to it the value of R (i.e., after the update of A into B, $ac^B(\pi_1^A, ..., \pi_n^A) = R^A$ must hold).

Example: the update $ac(3) := 1$ gives the value 1 to $ac(3)$; the value of ac is not changed elsewhere.

In the general case, the arguments π_i define a set of points where the function ac is updated: these points are computed by assigning all possible values to the variables in π_i. For the other values of the ac arguments, the result is not changed (it is the classical *frame assumption*).

Examples

- The update $\forall y\ [ac(y) := 0]$ forces ac to yield 0 for any argument.
- The update $\forall y\ [ac(succ(y)) := 1]$ assigns the value 1 to $ac(y)$ for all $y \neq 0$ and let the value of $ac(0)$ unchanged.

The right-hand side argument, R, of an elementary modifier is a term of sort s composed over the π_i, and defining the new results of ac at the update points. This ensures that an assignment of the variables in π_i uniquely defines the value of R. Counter-examples justifying this restriction are given in [9].

Example: the update $\forall y\ [ac(s(y)) := s(s(y))]$ assigns the value $y + 1$ to $ac(y)$ for all $y \neq 0$ and leaves the value of $ac(0)$ unchanged.

4.3 Conditional Elementary Modifiers

A conditional elementary modifier has the following form:

$\forall y_1, \ldots, y_p$ **cases**
ϕ_1 **then** $ac(\pi_1^1, \ldots, \pi_n^1) := R^1 | \ldots | \ \phi_m$ **then** $ac(\pi_1^m, \ldots, \pi_n^m) := R^m$
end cases

It describes a modification of the same elementary access function, ac, which is different depending on the different validity domains of the ϕ_i. In case of conflicts, i.e., if several ϕ_i are simultaneously valid, the update corresponding to the smallest index takes place.

Example: Let us have the access functions $ac_1, ac_2 : Nat \rightarrow Nat$ and the following operations:

$f_1, f_2 : Nat \rightarrow Nat$
$null : Nat \rightarrow Bool.$
The following conditional elementary modifier:
$\forall n$ **cases**
$null(ac_1(n)) = true$ **then** $ac_2(n) := f_1(n)$ |
$null(ac_1(n)) = false$ **then** $ac_2(n) := f_2(n)$
end cases

assigns to ac_2 the value $f_2(n)$, for all n, when the corresponding value of $ac_1(n)$ does not satisfy the condition $null$ and the value $f_1(n)$ in the opposite case.

Like in elementary modifiers we have: $\pi_1^i, \ldots, \pi_n^i \in T_{\Sigma''}(x_1,\ldots,x_q,y_1 \ldots y_p)_{s_j}$, where $(x_1, \ldots, x_q)$ are the parameters of the modifier being defined, and R_i is a Σ'' term built over $\pi_1^i, \ldots, \pi_n^i$. The form of the conditions ϕ_i depends on the underlying data type specification language. The terms in the conditions ϕ_i, like the right hand side arguments R^i, belong to $T_{\Sigma''}(\pi_1^i, \ldots, \pi_n^i)$.

The main reason for the introduction of conditional elementary modifiers is the possibility of using in the conditions some variables in addition to the parameters of the modifier being defined. These variables are universally quantified like variables in patterns.

Example

Mod(x) = $\forall n$ **cases**
null($ac_1(n)$) = x **then** $ac_2(n) := f_1(n)$ |
null($ac_1(n)$) = *not*(x) **then** $ac_2(n) := f_2(n)$
end cases

In this example, the variable n, unlike the parameter x, is universally quantified and, therefore, the modification is performed for a given x (given by the invocation of the modifier Mod) and for all n.

4.4 Composed Update Expressions

Several forms of update expression composition are proposed.

- Conditional updates of the following form:
  begin ϕ_1 **then** $Em_1 | \ldots | \phi_p$ **then** Em_p **end**
  indicating that an update expression Em_i is chosen if its condition ϕ_i is valid. If several conditions ϕ_i are valid, the update expression with the smallest index is chosen.
  Note: This form of update is different from the conditional elementary modifier in two ways: the Em_i are any update expressions; there are no universally quantified variables.
- $m_1; m_2$ indicating that the execution of m_1 should be followed by that of m_2.
- m_1 **and** m_2 indicating that the order of execution of m_1 and m_2 is unimportant. It is the specifier's responsability to ensure that the same result will be produced in any order of execution. A sufficient but not necessary condition for this is that m_1 (resp. m_2) does not update an elementary access function used or updated by m_2 (resp. m_1).
  This composition is generalized for n update expressions, with the same responsability for the specifier: all permutations must lead to the same result.
- $m_1 \bullet m_2$ indicating that the updates specified by m_1 and m_2 should be applied to the same state. If m_1 and m_2 specify the update of the same access function (their sets of updated elementary access functions are not disjoint), each of them must update it at different points; otherwise, the update m_1 is taken into account. This composition is generalized for n update expressions.

Each of the composition operators **and** and $\bullet$ has its own purpose: the composition by **and** lets some liberty to the implementor. The specifier uses **and** to

indicate that the order is unimportant. The composition by • gives the specifier a greater expression facility by removing the need to care for intermediate results or value preserving in the specification. Examples are given in [9].

5 Semantics of Update Expressions

The semantics of an update is a transformation of a $< \Sigma', Ax >$-algebra into another one, respecting the partitioning of $< \Sigma', Ax >$-algebras into $state_A(\Sigma', Ax)$, as mentioned in Section 2.

To give the semantics of different update expressions, we first give the semantics of the basic update expressions, namely *elementary modifiers* and *conditional elementary modifiers*. On the basis of the semantics of these expressions, the semantics of composed update expressions and defined modifiers is then given.

We denote by $\overline{ass}$ the extension to $T_{\Sigma''}(X)$ of assignment ass ($ass : X \to A$, $ass = \{ass_s : X_s \to A_s | s \in S\}$, $\overline{ass} : T_{\Sigma''}(X) \to A$).

We denote by $[\![m]\!]$ the transformation associated with an update expression m. It respects the partitioning of $< \Sigma', Ax >$-algebras into $state_A(\Sigma', Ax)$, i.e.:

$$\forall A \in Alg(\Sigma, Ax),\ [\![m]\!] : state_A(\Sigma', Ax) \to state_A(\Sigma', Ax)$$

For instance, the semantics of **nil** is the simplest one, since no update is produced: $[\![\mathbf{nil}]\!]A' = A'$

5.1 Semantics of Elementary Modifiers

The definition of the semantics of an elementary modifier is

$[\![\forall(x_1 \dots x_p)[ac(\pi_1, \dots, \pi_n) := R]]\!]A' = F(A')$

where F is the total map on the class of Σ'-algebras which transforms a Σ'-algebra A' into a Σ'-algebra B' by replacing $ac^{A'}$ with $ac^{B'}$ which is defined below.

$\forall v_1, \dots, v_n \in A'_{s_1}, \dots A'_{s_n}$,

- if there exists an assignment $ass : \{x_1, ..., x_p\} \to A'$, such that $v_1 = \overline{ass}\pi_1, ..., v_n = \overline{ass}\pi_n$ and $v = \overline{ass}R$
- then $ac^{B'}(v_1, ..., v_n) = v$
- otherwise, $ac^{B'}(v_1, ..., v_n) = ac^{A'}(v_1, ..., v_n)$.

5.2 Semantics of Conditional Elementary Modifiers

The definition of the semantics of a conditional elementary modifier is:

$[\![\forall y_1, \dots, y_p$ **cases**
ϕ_1 **then** $ac(\pi_1^1, \dots, \pi_n^1) := R^1 | \dots |$
ϕ_m **then** $ac(\pi_1^m, \dots, \pi_n^m) := R^m$
end cases $]\!]A' = F'(A')$

where F' is the total map on the class of Σ'-algebras transforming a Σ'-algebra A' into a Σ'-algebra B' by replacing $ac^{A'}$ with $ac^{B'}$ in the following way.

$\forall v_1, \ldots, v_n \in A'_{s_1}, \ldots A'_{s_n}$,
- if there is no i such that there is an assignement ass: $\{y_1, \ldots, y_p\} \rightarrow A'$ with
  - $v_1 = \overline{ass}\pi^i_1, ..., v_n = \overline{ass}\pi^i_n$,
  and
  - ϕ_i is valid for this assignment in $Ext_{\Sigma''}(A')$ with the conventional interpretation of logical connectors,
- then $ac^{B'}(v_1, ..., v_n) = ac^{A'}(v_1, ..., v_n)$;
- otherwise, let I be the set of i satisfying the condition above, $j = min(I)$ and $v = \overline{ass}R^j$, then $ac^{B'}(v_1, ..., v_n) = v$

5.3 Semantics of Composed Update Expressions

– Let U be a conditional update of the form:
begin ϕ_1 **then** $Em_1 | \ldots | \phi_p$ **then** Em_p **end**
and let I be the set of i, such that $Ext_{\Sigma''}(A') \models \phi_i$. Then:
- If $I = \emptyset$, then $[\![U]\!]A' = A'$.
- Otherwise, $[\![U]\!]A' = [\![Em_{min(I)}]\!]A'$.

– $[\![m_1; m_2]\!]A' = [\![m_2]\!]([\![m_1]\!]A')$

– $[\![m_1$ **and** $m_2]\!]A' = [\![m_2$ **and** $m_1]\!]A' = [\![m_1; m_2]\!]A' = [\![m_2; m_1]\!]A'$
This definition is generalized for n arguments. The semantics of m_1 **and** ... **and** m_n is that of all the permutations of $m_1, \ldots m_n$.

– $[\![m_1 \bullet m_2]\!]A' \equiv G(A')$
where G is a total map transforming a Σ'-algebra A' into a Σ'-algebra B' by replacing each $ac^{A'}$ by $ac^{B'}$ as follows.
Let $[\![m_1]\!]A' = A1$ and $[\![m_2]\!]A' = A2$. Then G transforms A' into a Σ'-algebra B' by replacing, for each operation name $ac : s_1, ..., s_n \rightarrow s$ in Σ_{eac} and each value v_i of sort s_i, $ac^{A'}$ by $ac^{B'}$ in the following way:
- if $(ac^{A'}(v_1, ..., v_n) = ac^{A1}(v_1, ..., v_n)) \wedge (ac^{A'}(v_1, ..., v_n) = ac^{A2}(v_1, ..., v_n))$ then $ac^{B'}(v_1, ..., v_n) = ac^{A'}(v_1, ..., v_n)$ (there is no update at this point);
- if $(ac^{A'}(v_1, ..., v_n) \neq ac^{A1}(v_1, ..., v_n)) \wedge (ac^{A'}(v_1, ..., v_n) = ac^{A2}(v_1, ..., v_n))$ then $ac^{B'}(v_1, ..., v_n) = ac^{A1}(v_1, ..., v_n)$(the update comes from $A1$);
- if $(ac^{A'}(v_1, ..., v_n) = ac^{A1}(v_1, ..., v_n)) \wedge (ac^{A'}(v_1, ..., v_n) \neq ac^{A2}(v_1, ..., v_n))$ then $ac^{B'}(v_1, ..., v_n) = ac^{A2}(v_1, ..., v_n)$ (the update comes from $A2$);
- if $(ac^{A'}(v_1, ..., v_n) \neq ac^{A1}(v_1, ..., v_n)) \wedge (ac^{A'}(v_1, ..., v_n) \neq ac^{A2}(v_1, ..., v_n))$ then $ac^{B'}(v_1, ..., v_n) = ac^{A1}(v_1, ..., v_n)$ (both m_1 and m_2 update the same access function at the same point, the first update is taken into account).

This definition is generalized for n arguments.

5.4 Semantics of the Definition and Invocation of Defined Modifiers

Let $mod(x_1, ..., x_n) = Em$ be a modifier definition. Then, for any $D(A)$-algebra A' and ground Σ'' terms $t_1, ..., t_n$ of sorts $s_1, \ldots, s_n$ respectively, the map $mod^{D(A)}$ associated with mod in a dynamic system $D(A)$ is defined as:

$$mod^{D(A)}(< A', < t_1^{A'}, ..., t_n^{A'} >>) = [\![Em[t_1/x_1, ...t_n/x_n]]\!]A',$$

where $Em[t_1/x_1, ...t_n/x_n]$ is the update expression obtained by replacing each x_i in Em by t_i.

Thus the semantics of an invocation $mod(t_1, \ldots, t_n)$ is:

$$[\![mod(t_1, ..., t_n)]\!]A' = mod^{D(A)}(< A', < t_1^{A'}, ..., t_n^{A'} >>)$$

6 States and Behaviors of the System

We summarize in this section the main definitions related to the notions of state and behavior of a dynamic system.

Let $DS = < (\Sigma, Ax), (\Sigma_{eac}, Ax_{Init}), (\Sigma_{ac}, Ax_{ac}, \Sigma_{mod}, Def_{mod}) >$ be a specification of a dynamic system, and $\Sigma' = \Sigma \cup \Sigma_{eac}$.

System's state. As already mentioned, a state of the system, defined by the specification DS is a Σ'-algebra satisfying the axioms Ax.

Initial states. A subset of the set of states represents the possible initial states of the specified system. It corresponds to an enrichment of the specification $< \Sigma', Ax >$ with Ax_{Init} , thus:

$$state_{Init}(DS) = \{A' \in Alg(\Sigma', Ax >) | A' \models Ax_{Init}\}$$

Behavior of the system. A behavior is a sequence of updates which are produced by the invocations of some defined modifiers. Several sequences of states $(e_0, e_1, e_2, ...)$ correspond to a behavior $(m_0, m_1, m_2, ...)$ depending on the choice of the initial state:

- the initial state e_0 belongs to $state_{Init}(DS)$;
- each e_{i+1} is the result of the application of the modifier m_i to e_i $(e_{i+1} = [\![m]\!]e_i)$.

The semantics of updates as it is defined in the previous section guarantees that if e_0 belongs to a dynamic system $D(A)$, then any e_i also belongs to $D(A)$ (the state changes, but the data types do not change).

This formalism is deterministic for two reasons: the semantics of elementary modifiers and, therefore, of all modifiers ensures that only one state (up to isomorphism) is associated with the application of a modifier to a state; besides the specification of dependent access functions, $< \Sigma_{ac}, Ax_{ac} >$, is sufficiently complete with respect to $< \Sigma \cup \Sigma_{eac}, Ax >$ (cf. Section 3). Thus, only one sequence of states starting with a given initial state is associated with a behavior.

Reachable states. The set of reachable states, $REACH(DS)$ is the set of states which can be obtained by a sequence of updates corresponding to the invocations of some modifiers of Σ_{mod}, starting from an initial state.

Thus, the set $REACH(DS)$ is recursively defined in the following way:

- $state_{Init}(DS) \subset REACH(DS)$
- $\forall m \in \Sigma_{mod}, \forall t_1 \in (T_{\Sigma''})_{s_1} \ldots t_n \in (T_{\Sigma''})_{s_n}, \forall A' \in REACH(DS)$, $[\![m(t_1, ..., t_n)]\!]A' \in REACH(DS)$.

6.1 Properties and Invariants

To prove that a property F is valid in any extended state of a dynamic system $D(A)$, i.e, that:

$$\forall A' \in |D(A)|, Ext_{\Sigma''}(A') \models F,$$

one can use the logic and tools of the underlying algebraic specification language.

An invariant is a property, *Inv*, which must be valid in all reachable states:

$$\forall A' \in REACH(DS), Ext_{\Sigma''}(A') \models Inv$$

Example: $max \geq delay$ is an invariant in the example of Figure 1.

To verify an invariant Inv, one can proceed by induction on the reachable states. First, it must be proved that Inv holds in the initial states. To do this, it is sufficient to use the logic and tools of the algebraic specification language to prove that Inv is a consequence of $Ax \cup Ax_{Init}$. Then it must be proved that the application of each modifier preserves Inv.

Currently, there is no formal calculus for the modifiers definitions. Therefore, the demonstration of the following properties must be done on the basis of the definitions of the semantics, i.e on the properties of the elementary modifiers and their compositions given in the previous section:

$\forall A' \in D(A), \forall mod \in \Sigma_{mod}, \forall t_1 \in (T_{\Sigma''})_{s_1} \ldots t_n \in (T_{\Sigma''})_{s_n}$
$Ext_{\Sigma''}(A') \models Inv => Ext_{\Sigma''}([\![mod(t_1 \ldots t_n)]\!]A') \models Inv$

7 Related Works

"Evolving algebras", also called "Abstract State Machines", have been proposed by Gurevich [14] and then intensively used for formal definition of various algorithms and programming language semantics. They are based on the notion of a universal algebraic structure consisting of a set (superuniverse), a number of functions, and a number of relations. Data types (universes) are modeled by unary relations on the superuniverse. Functions can be either static or dynamic. A static function never changes, a change of a dynamic function produces a new algebra. Another means of algebra modification is changing the number of elements in the underlying set (importing new elements).

When writing a specification with the use of a conventional Abstract State Machines, one can write the signature of any function operating with values of one or more universes. One cannot, however, define formally the semantics of a static function as an abstract data type. As a result, one gets a specification where a number of data types and functions are introduced informally (one can make sure of this, looking at the definition of C [15] where almost all static functions and data types are defined in plain words). Besides, there is no notion similar to those of dependent access functions and modifiers with patterns present in our approach.

"Dynamic abstract types" are informally introduced in [6] as a wishable general framework for specification. It is proposed that such a type should consist

of an abstract data type and a collection of dynamic operations. Four levels of specification are outlined: value type specification, instant structure specification, dynamic operation specification, and higher-level specification. Access functions and modifiers, as defined here, are just dynamic operations, and the specification technique proposed in our paper can be used for this type of specification.

An idea similar to our state-as-algebra approach is proposed in terms of a new mathematical structure, called "d-oid", by Astesiano and Zucca [1]. A d-oid, like our dynamic system, is a set of instant structures (e.g., algebras) and a set of dynamic operations (transformations of instant structures with a possible result of a definite sort). Here dynamic operations serve as counterparts of our access functions and modifiers. However, the approach in question deals only with models and does not address the issue of specifying the class of such behaviors, which is our focus.

The idea of dynamic types is also investigated in [26]. Although no direct definition of a dynamic abstract type is given in that paper, it has contributed by formal definitions of a static framework and of a dynamic framework over a given static framework. We have used the idea of dynamic operations to define the semantics of our modifiers, and we propose in addition an approach to their formal specification.

Another similar approach is the "Concurrent State Transformation on Abstract Data Types" presented by Grosse-Rode in [12] and recently revised in [13] as "Algebra Transformation Systems". States are modeled as partial algebras that extend a fixed partial algebra considered as a static data type. All functions are given at the same level. Dynamic functions are considered totally undefined in the static data type. A state on a given partial algebra is a free extension of this algebra, specified by a set of function entries. Invariant relations between dynamic operations are given by axioms at the static level. Transitions between states are specified by conditional replacement rules. A replacement rule specifies the function entries that should be added/removed when the condition is valid.

There are some restrictions on the partial equational specifications for the static data types, the admissible partial algebras and states, and the replacement rules in order to have the same structural properties as the algebraic specification logic. A problematic issue is the checks that the replacement rules are compatible with the axioms. This leads to severe restrictions on the use of the formalism. We do not have this problem because the axioms of the data types are clearly isolated, and, moreover, we don't consider the axioms on dependent accesses in the state. In [13] the semantics is revised and a theoretical framework for the composition of these algebra transformation systems is given.

The "Hidden Sorted Algebra" approach [11], where some sorts are distinguished as hidden and some other as visible, treats states as values of hidden sorts. Visible sorts are used to represent values which can be observed in a given

state. States are explicitly described in the specification in contrast to our approach.

The above work combined with Meseguer's rewriting logic [19] has served as basis of the dynamic aspects of the CafeOBJ language [4]. There, states and transitions are modeled, respectively, as objects and arrows belonging to the same rewrite model which is a categorical extension of the algebraic structure. Meseguer's rewriting logic is also basis of the specification language Maude [20].

Another approach to the formalization of object behaviors is the concept of "Coalgebra" presented in [21]. Each object state is represented as an element of a special set of a coalgebra, with a notion of equality of object behaviors which is close to the behavioral equivalence defined for hidden sorted algebras [11].

Our framework is different of the three ones above, since we consider states as algebras, not only as elements of an algebra. It avoid the specification of the, often complex, data type corresponding to the state.

Finally, the specification language Troll [16] should be mentioned. Troll is oriented to the specification of objects where a method (event) is specified by means of evaluation rules similar to equations on attributes. Although the semantics of Troll is given rather informally, there is a strong mathematical foundation of its dialect Troll-light [10], with the use of data algebras, attribute algebras and event algebras. A relation constructed on two sets of attribute algebras and a set of event algebra, called *object community*, formalizes transitions from one attribute algebra into another when a particular event algebra takes place.

8 Conclusion

In this paper we have presented a specification method based on the concept of implicit state by giving some syntax and its semantics. This approach is based on the *algebras-as-states* paradigm which has been recently re-explored by several authors. Our proposal aims at combining the advantages of these works with a strong motivation to keep the specifications as abstract and non algorithmic as possible. This is achieved via several means.

Our specifications of dynamic systems are extensions of some algebraic specification of data types which gives a formal and abstract definition of these types. As said above, it is an advantage w.r.t. Abstract State Machines. It is also an advantage w.r.t. approaches such as VDM or Z since data types are defined independently of any predefined library.

The notions of elementary accesses and dependent accesses makes it possible to describe behaviors where an internal memory evolves, without deciding at the specification level what will be stored or computed by the implementation. The notion of dependent access has been designed to provide a convenient means for describing abstractly states where several related values evolve: such accesses can be implemented either by some memory locations or by some functions, depending on efficiency considerations.

The powerful concepts of patterns in modifier definitions and conditional elementary modifiers make it possible to specify global changes of the implicit

state in a non algorithmic way. It avoids the use of loops and iterations and provides a black-box way of specifying complex modifications of the state.

The fact that elementary modifiers are hidden ensures encapsulation.

This framework has been validated on several case studies and is easy to learn and use. We plan to use it as a basis for an algebraic specification language of object oriented systems. Such a language should allow the description of systems where several, named, encapsulated, implicit states coexist and communicate, appear and disappear, independently of any specific object oriented programming approach.

Acknowledgement

We warmly thank Pierre Dauchy for his numerous and important contributions to AS-IS.

References

1. E. Astesiano and E. Zucca. D-oids: a model for dynamic data types. *Mathematical Structures in Computer Science*, 5(2):257–282, 1995. 125
2. P. Dauchy. Développement et exploitation d'une spécification algébrique du logiciel embarqué d'un métro. Thèse, Université de Paris-Sud, Orsay, 1992. 114
3. P. Dauchy and M.-C. Gaudel. Algebraic specifications with implicit state. Rapport interne 887, Laboratoire de Recherche en Informatique, February 1994. 114
4. R. Diaconescu and K. Futatsugi. *CafeOBJ Report*, volume 6 of *AMAST series in Computing*. World Scientific Publishing Co. Pte. Ltd, 1998. 126
5. H. Ehrig and B. Mahr. *Fundamentals of algebraic specification,Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer science*. Springer-Verlag, 1985. 117
6. H. Ehrig and F. Orejas. Dynamic abstract data types : An informal proposal. In *Bull. of EATCS*, volume 53, pages 162–169, 1994. 124
7. M.-C. Gaudel. *Algebraic Specifications*, chapter 22. Software Engineer's Reference Book, McDermid, J., ed. Butterworths, 1991. 117
8. M.-C. Gaudel, P. Dauchy, and C. Khoury. A formal specification of the steam-boiler control problem by algebraic specifications with implicit state. In *Formal Methods for Industrial Applications, Specifiying and Programming the Steam Boiler Control*, volume 1165 of *LNCS*, 1996. 114
9. M.-C. Gaudel, C. Khoury, and A. Zamulin. Dynamic systems with implicit state. Rapport interne 1172, Laboratoire de Recherche en Informatique, May 1998. 118, 119, 121
10. M. Gogolla and R. Herzig. An algebraic semantics for the object specification language TROLL-light. In *Recent Trends in Data Type Specifications*, volume 906 of *LNCS*, pages 290–306, 1995. 126
11. J. Goguen and R. Diaconescu. Towards an algebraic semantics for the object paradigm. In *Recent Trends in Data Type Specifications*, volume 785 of *LNCS*, 1994. 125, 126

12. M. Grosse-Rhode. Concurrent state transformation on abstract data types. In *Recent Trends in Data Type Specification*, volume 1130 of *LNCS*, pages 222–236, 1996. 125
13. M. Grosse-Rhode. Algebra transformation systems and their composition. In *Fundamental Approaches to Software Engineering*, volume 1382 of *LNCS*, pages 107–122, 1998. 125
14. Y. Gurevich. *Evolving Algebras 1993: Lipari Guide, In Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995. 114, 124
15. Y. Gurevich and J. Huggins. The semantics of the C programming language. In *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309, 1993. 124
16. T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kush. Revised version of the modelling language TROLL. Technical Report 03, Technishe Universitaet Braunschweig, Informatik-Berichte, 1994. 126
17. C.-B. Jones. *Systematic Software Development using VDM*. Prentice Hall International series in computer science, 1989. 115
18. C. Khoury, M.-C. Gaudel, and P. Dauchy. AS-IS. Rapport interne 1119, Laboratoire de Recherche en Informatique, August 1997. 114, 115
19. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, April 1992. 126
20. J. Meseguer and Winkler. Parallel programming in Maude. In *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *LNCS*, pages 253–293, 1992. 126
21. H. Reichel. An approach to object semantics based on terminal coalgebras. In *Recent Trends in Data Type Specifications*, volume 906 of *LNCS*, pages 129–152, 1995. 126
22. J.-M. Spivey. *The Z Notation, A reference manual*. Prentice Hall International series in computer science, 1987. 115
23. A.-V. Zamulin. Specification of an Oberon compiler by means of a typed gurevich machine. Technical Report 58939450090000701, Institute of Informatics Systems of the Siberian Division of the Russian Academy of Sciences, Novosibirsk, 1996. 114
24. A.-V. Zamulin. Typed Gurevich Machines. Technical Report 36, Institute of Informatics Systems, Novosibirsk, 1996. 114
25. A.-V. Zamulin. Typed Gurevich Machines Revisited. In *Joint NCC&ISS Bulletin*, volume 7 of *Comp. Science*, pages 95–121, 1997. 114, 115
26. E. Zucca. From static to dynamic data-types. In *Mathematical Foundations of Computer Science*, volume 1113 of *LNCS*, pages 579–590, 1996. 125