Joint NCC & IIS Bull., Comp. Science, 8 (1998), 101–127
© 1998 NCC Publisher

Object-Oriented Specification by Typed Gurevich Machines*

A.V. Zamulin

An approach to the object-oriented specification by typed Gurevich machines is proposed in the paper. The approach is based on considering an object update as a transition from one algebra of a given signature to another of the same signature. Each object possesses a state and a unique identifier; the state of a mutable object can be updated, the state of a constant object cannot be updated. An algebra provides two sets of unique identifiers for each object type: a set of mutable object identifiers and a set of constant object identifiers. An object type signature introduces the observers of the corresponding set of objects serving to inspect object states and mutators serving to initialize and update observers of mutable objects. Transition rules of a typed Curevich machine are proposed as a means of object specification. The specification methodology is quite straightforward: an object initialization or modification by means of a mutator is expressed in terms of updates of the object's observers.

1. Introduction

The aim of this paper is to extend and adapt the mechanism of typed Gurevich machines [1] to provide an object-oriented style of the specification of a complex dynamic system. Following [2], we understand an object as a complex entity having a unique identifier and state which changes during the object's life. An object's state can be initialized by a number of *intitializers*, updated by a number of *mutators* and viewed by a number of *observers* (which are instances of *methods*). Objects can be created as soon as they are needed; they can be shared, passed as function parameters, compared for equality and sameness. In addition to mutable (variable) objects whose state can evolve with time, constant (immutable) objects which retain their state until they are destroyed should be allowed. An object can belong to several object types according to the IS-A relationship. Thus, we follow the typical paradigm of object-oriented imperative languages like C^{++} in contrast to approaches treating objects from a purely functional point of view [3].

The motivation of this work stems from the author's dissatisfaction with

*Partially supported by the Russian Foundation for Basic Research under Grant 98–01–00682.

some attempts to give a classical algebraic foundation of the object-oriented programming paradigm [4, 5, 6]. In contrast to a value of a conventional data type, an object is generally a dynamic entity whose state evolves with time. Although the author's experience with formal definition of the programming language Oberon [7] has shown that the state can be modeled by a suitable data type in the framework of conventional algebraic specifications, this experience has also shown that such a modeling is quite artificial and tedious. A similar observation is noted in [8]. Therefore, a technique for modeling dynamic systems is called for.

The approach followed in the paper is mainly based on Gurevich's work on abstract state machines [9, 10] and effected by the works on algebraic specifications with implicit state [8, 11], d-oids [12], dynamic abstract types [13]. The paper is organized in the following way. Type-structured specifications representing the static part of a dynamic system are informally introduced in Section 2. Object-structured signatures and relationships among them are defined in Sections 3 and 4, respectively. Object-structured algebras are introduced in Section 5. The construction and interpretation of terms of an object-structured signature are discussed in Section 6. Transition rules serving as a tool for object type specification are defined in Section 7, and object-structured specifications themselves are defined in Sections 8 and 9. Some related work is discussed in Section 10, and some conclusions and directions of further work are outlined in Section 11.

2. Type-structured specifications

The specification of any state (instance algebra) of a dynamic system is a typical many-sorted algebraic specification. Therefore, any specification language whose semantics is a class of many-sorted algebras is suitable for our purpose. However, to make the specifications of static entities (viz. data types) and dynamic entities (viz. objects) as similar as possible, we prefer to work with type-structured algebraic specifications.[2]

Let $\Sigma = \langle TYPE, \Omega \rangle$, where $TYPE$ is a set of (data type) names and Ω is a set of function symbols each indexed with a function profile (u, s), such that $u \in TYPE^*$ and $s \in TYPE$, be a many-sorted signature. We consider Σ to be structured in such a way that each name $T \in TYPE$ is associated with a set of function symbols $\omega_{us} \in \Omega$, so that each ω_{us} in this set uses T as an element (elements) of u and/or as s. We call such a signature a *type-structured signature*, and we call an association of a data type name and a set of pertinent function symbols a *data type signature*.

An algebra of a type-structured signature is built in the conventional way

[2]Fundamentals of structured specifications can be found in [14]; type-structured specifications are formally introduced in [15].

by associating sets of elements with names in $TYPE$ and (partial) functions with function symbols in Ω. Terms are also built conventionally. If A is a Σ-algebra, then $|A|$ is the carrier of A and A_T is the set of elements associated in A with the name $T \in TYPE$.

A data type signature supplied with a set of axioms is a *data type specification*. To express the definedness of a term of a type-structured specification, we use a special semantic predicate D, such that $D(t)$ states that the term t is defined in an algebra A iff the interpretation of the term, t^A, produces some object in A.

Example.

type SeqNat = **spec**
[empty: SeqNat;
append: Nat, SeqNat $\longrightarrow$ SeqNat;
head: SeqNat $\longrightarrow$ Nat;
tail: SeqNat $\longrightarrow$ SeqOfNat;
has: SeqNat, Nat $\longrightarrow$ Boolean;
is_empty: SeqNat $\longrightarrow$ Boolean]
{**forall** x, y: Nat, s, s1: SeqNat.
dom head(s): ¬is_empty(s); **dom** tail(s): ¬is_empty(s);
is_empty(empty) == true; is_empty(append(x, s)) == false;
head(append(x, s)) == x; has(empty, x) == false;
has(append(y, s), x) == x = y | has(s, x); tail(append(x, s)) == s}.

Notation. In the above specification, the data type signature is enclosed in square brackets, and the set of axioms is enclosed in curly brackets (two parts of an axiom are equated by the symbol "=="); the clause **dom** specifies the domain of a partial function: in a domain specification **dom** $t : b$, $D(t)$ holds if and only if b evaluates to *true*; for example, *head*(s) is defined only if s is not empty.

We assume in the sequel that any specification includes the conventional specifications of the data types Nat and $Boolean$.

3. Object-structured signature

Let $\Sigma' = \langle TYPE, \Omega\rangle$, where $TYPE$ is a set of sort (type) names and Ω is a set of operators indexed with operation profiles constructed with the use of names from $TYPE$, be a many-sorted signature. An *object-structured signature* over Σ' is defined in the following way. Let

- $OTYPE$ be a set of names (of object types) such that $TYPE \cap OTYPE = \emptyset$;

- $OTYPE^\circ$ be an extension of $OTYPE$ created in the following way: if $T \in OTYPE$, then $T, Var(T), Const(T) \in OTYPE^\circ$;
- an *observer profile* be either T or $T_1, \ldots, T_n \longrightarrow T$, where $T, T_i \in TYPE \cup OTYPE^\circ, i = 1, \ldots, n$;
- a *transformer profile* be either $T_1, \ldots, T_n$ or $T_1, \ldots, T_n \longrightarrow T$ or $\longrightarrow T$, where $T, T_i \in TYPE \cup OTYPE^\circ, i = 1, \ldots, n$;
- an *attribute signature* be a pair $at : T$, where at is a name and $T \in TYPE \cup OTYPE^\circ$;
- an *observer signature* be a pair $b : OP$, where b is a name and OP is an observer profile;
- a *transformer signature* be either m or $m : MP$, where m is a name and MP is a transformer profile.

Then an *object type signature* is a set of attribute signatures, observer signatures, and transformer signatures (called *component signatures* in the sequel).[3] An *object-structured signature* Σ is a tuple $\langle \Sigma', OTYPE, O\Phi, int^o \rangle$, where $O\Phi$ is a set of object type signatures and int^o is a function mapping $OTYPE$ into $O\Phi$. For any $T \in OTYPE$ and $o\phi \in O\Phi$, we say that $o\phi$ is marked with T if $int^o(T) = o\phi$.

Notation: we introduce an object type signature with a keyword **class**, sets of attributes, observers, and transformers are preceded by keywords **attribute**, **observer**, and **transformer**, respectively. The function int^o is represented as a set of pairs *object type name = object type signature.*

The following examples introduce the signatures of a recursive object type and several tuple-structured object types (note the difference in the signatures of the data type "SeqNat" and object type "SeqOfNat"):

```
class SeqOfNat = spec
[transformer  empty; – construction of an empty sequence
              append: Nat; – appending a natural number to a sequence
              delete_head: ⟶ Nat; – deleting the head of a sequence
 attribute head: Nat; – fetching the head of a sequence
              tail: SeqOfNat; – fetching the tail of a sequence
              is_empty: Boolean; – checking whether a sequence is empty;
 observer has: Nat ⟶ Boolean; – checking for the presence of an element

class Date = spec
[transformer create_date: Nat, Nat, Nat; – creating a date
 attribute day, month, year: Nat] – fetching the value of day, month or year;
```

[3]Note that the technique of profile definitions here more resembles the corresponding technique of object-oriented programming languages than that of algebraic specification languages; this is natural since an object is not a value.

```
class Person = spec
 [transformer create_person: String, Const(Date);
            marry: Var(Person);
            divorce;
  attribute name: String;
            spouse: Var(Person);
            birth_date: Const(Date)];

class Rectangle = spec
 [transformer default_rectangle; – default rectangle
            create: Nat, Nat; – creating a new rectangle
  attribute length, width: Nat – getting rectangle parameters;
  observer area: Nat – computing the rectangle's area;
            equal: Rectangle ⟶ Boolean] – comparison for equality.
```

From the intuitive point of view, a tuple of attributes defines an object's state, an observer is a function computing something at a given object's state, and a transformer is a procedure creating or changing an object's state. Attributes are often called *instance variables* and observers and transformers are often called *methods* in programming languages.

4. Relationships between object type signatures

Let us consider the following object types:

```
class Point = spec
 [attribute x, y: Int;
  transformer create_point: Int, Int;
            default_point;
            move: Int, Int;
  observer get_x: Int;
            get_y: Int];

class Colorpoint = spec
 [attribute x, y: Int;
            color: Color;
  transformer create_point: Int, Int, Color;
            default_point;
            set_color: Color;
            move: Int, Int;
  observer get_x: Int;
            get_y: Int;
            get_color: Color].
```

We can see that the signature of *Colorpoint* practically inherits the signature of *Point* and adds some extra attributes, transformers and observers. It is intuitively clear that anywhere a point object is needed, a colorpoint object can be used. Therefore, a supertype-subtype relation between these two object types can be set.

It is tempting to state that an object type T_1 is a subtype of an object type T if the signature of T is included in the signature of T_1. However, this is not the case. There is a small difference in the profiles of the transformers *create_point*. Intuitively, it is clear that *create_point* does not refer to "proper" transformers it actually refers to initializers used to set some initial values in a point or colorpoint object. Since a colorpoint object has an extra attribute, the transformer has an extra parameter.

To overcome the problem, we divide the set of transformer signatures in two sets, *initializer signatures* and *mutator signatures* and rewrite the previous two examples in the following way:

```
class Point = spec
 [attribute x, y: Int;
  intitializer create_point: Int, Int;
  mutator default_point;
          move: Int, Int;
  observer get_x: Int;
          get_y: Int];

class Colorpoint = spec
 [attribute x, y: Int;
          color: Color;
  intitializer create_point: Int, Int, Color;
  mutator default_point;
          set_color: Color;
          move: Int, Int;
  observer get_x: Int;
          get_y: Int;
          get_color: Color].
```

We can now clearly see that the signature of *Point* is included in the signature of *Colorpoint* if initializers are not considered[4].

Definition. Let *Att*(T), *Obs*(T), and *Mut*(T) be, respectively, the sets of attribute, observer, and mutator signatures in the object type signature marked with T. Then an object type T_1 is a *subtype* of an object type T

[4]In fact, *default_point* are also intitialisers; however, we can include each of them in the set of mutators since they have the same profile.

(T is a ***supertype*** of T_1) iff $Att(T) \subseteq Att(T_1)$ and $Obs(T) \subseteq Obs(T_1)$ and $Mut(T) \subseteq Mut(T_1)$.

In this way, we want the subtype to have more components in comparison to the supertype; thus, a subtype object can be considered as a supertype object when needed. If T_1 is a subtype of T, we write $T_1 < T$ when we want to stress that T_1 and T are different, and we write $T_1 \leq T$ when we assume that both T_1 and T are the same type (an object type is its subtype by definition).

We do not put any restriction on the number of types a given object type can be a subtype of; thus, both single and multiple inheritance are possible. However, we put a restriction on component signatures to avoid clashes between several supertypes.

Definition. An object-structured signature is hierarchically-consistent if for any object type T there are no two supertypes T_1 and T_2 having the same component signature.

We consider only hierarchically-consistent signatures in the sequel. In a concrete specification language some special means could be provided to resolve clashes between several supertypes as it is done in some programming languages; the discussion of these means is out of the scope of the present paper.

Definition. An object type T is called a *root type* for an attribute $at : T_1', \ldots, T_n' \longrightarrow T'$ (at is respectively called a *root attribute* of T) if there is no T_1, such that $T < T_1$ and $(at : T_1', \ldots, T_n' \longrightarrow T') \in Att(T_1)$.

The subtype-supertype relationship as defined above allows us to construct an algebra where subtype objects have components (excluding initializers) sharing the same name and profile with the corresponding components of a supertype object. This gives us the possibility to regard a subtype object as supertype object while permittimg the use of the subtype methods (late binding).

5. Object-structured algebras

5.1. Instance algebras

An *instance* algebra represents some state of a number of objects. The update of an object's state as well as the creation of an object leads to the transformation of one instance algebra into another. Let A' be a Σ'-algebra called a ***static algebra*** (this algebra provides a set of data type implementations and static functions defined over them). An instance algebra, A, of the signature

$\Sigma = \langle \Sigma', OTYPE, O\Phi, int^o \rangle$ is built as an extension of A' in the following way:

1. A set of elements, $A_{Var(T)}$, and a set of elements, $A_{Const(T)}$, such that $A_{Var(T)} \cap A_{Const(T)} = \oslash$, are assigned to each $Var(T), Const(T) \in OTYPE^{\circ}$, respectively, so that if $T < T'$, then $A_{Var(T)} \subset A_{Var(T')}$ and $A_{Const(T)} \subset A_{Const(T')}$.
2. A set of elements $A_T = A_{Var(T)} \cup A_{Const(T)}$ is assigned to each $T \in OTYPE$; these elements are called (*mutable, constant*) *object identifiers*. Note that the set of object identifiers of a supertype contains object identifiers of all its subtypes.
3. A partial function $at^A_T : A_T \longrightarrow (A_{T_1}, \ldots, A_{T_n} \longrightarrow A_{T'})$ is associated with each root attribute name $at : T_1, \ldots, T_n \longrightarrow T'$ in an object type signature marked with T; such a function is called an *attribute function*. A non-root attribute name $at : T_1, \ldots, T_n \longrightarrow T'$ in an object type signature marked with T_1, such that $T_1 < T$, is mapped to the same function. In this case, $at^A_{T_1}$ is an alternative name of the function. One can say that a supertype attribute function is inherited in each of its subtypes. If $id \in A_T$, then $at^A_T(id)$ is an *attribute* of id.

Thus, an instance algebra is a kind of order-sorted algebra [16]. An *object* is a pair (id, obs) where id is an object identifier and obs is a tuple of its attributes called *object's state*. We write sometimes "object *id*" meaning an object with the identifier id. An object id is mutable if $id \in A_{Var(T)}$ and an object id is constant if $id \in A_{Const(T)}$. If $id \in A_{Var(T)}$, an update of an attribute of id leads to the change of its state.

For the object type *Rectangle* introduced above, an object identifier could be represented, for example, by the address of a location capable to store tuples of *length* and *width* values, the attribute functions $length^A_{Rectangle}$ and $width^A_{Rectangle}$ would map location addresses to natural numbers, the observer $area^A_{Rectangle}$ would compute the area of a rectangle, and the observer $equal^A_{Rectangle}$ would compare the contents of two locations for equality.

To define the interpretation of observer and mutator names, we need firstly introduce the notions of *dynamic system* and *update set.*

5.2. Dynamic system

We discussed above only functions defined inside the frames of data (object) types. In a more general case, an instance algebra can possess a number of "independent" functions and constants defined outside of a data type or object type frame. Such functions and constants are called *dynamic* and can be different in different instance algebras. Some procedures transforming one instance algebra into another by updating objects and dynamic functions (constants) can also be defined. Thus, we define a *dynamic system signature* as an extension of an object-structured signature: $D\Sigma = < \Sigma, DF >$, where Σ is an object-structured signature and DF is set of function[5] and procedure signatures defined in the same way as attribute and transformer signatures are, respectively, defined above.

Notation: we introduce dynamic functions and constants with the keyword **dynamic** and procedures with the keyword **proc**.

Examples.
dynamic a_point: Var(Point);
dynamic a_colpoint: Var(Colorpoint);
dynamic matrix: Nat, Nat $\longrightarrow$ Nat;
proc swap: Var(Person), Var(Person);

For any dynamic system signature $D\Sigma = < \Sigma, DF >$, a Σ-algebra A is extended by an element $c^A \in A_T$ associated with the constant signature $c : T$ from DF and a (partial) function $f^A : A_{T_1} \times \ldots \times A_{T_n} \longrightarrow A_T$ associated with the function signature $f : T_1, \ldots, T_n \longrightarrow T$ from DF. Terms constructed with the use of constant or function names are interpreted by invocations of the corresponding constants or functions. In the sequel, mentioning an instance algebra A, we mean a $D\Sigma$-algebra.

Definition. Let OID be a set of object identifiers, and $|D(A')|$ be a set of instance algebras satisfying the following conditions: (i) all algebras in $|D(A')|$ have the same static algebra A', and (ii) if $T \in OTYPE$ and A and B are two $D\Sigma$-algebras, then both A_T and B_T are subsets of OID (i.e., objects identifiers are always chosen from the same set). Then an object-oriented dynamic system $D(A')$ of signature $D\Sigma$ consists of OID, $|D(A')|$, a set of algebra modifiers (defined below), and a set of observers, transformers, and procedures (defined below).

Note that different instance algebras of the same $D(A')$ can generally have different sets of object identifiers and different sets of attribute functions, which means that the sets of objects can be different and/or an object with the same identifier can have different states. At the same time, data type implementations and static functions are the same in all instance algebras of the same $D(A')$.

5.3. Function updates

One instance algebra can be transformed into another by *algebra modifiers*. Two algebra modifiers serve for algebra transformation by means of

[5] A function without arguments is called a constant in the sequel.

function updates. Let algebra A have a partial function with the signature $f : T_1, \ldots, T_n \longrightarrow T$.

Definition. A function update α in A is a triple $(f, \langle a_1, \ldots, a_n\rangle, a)$, where $a \in A_T, a_i \in A_{T_i}, i = 1, \ldots, n$. The modifier $\mu 1$ applied in A to an $\alpha = (f, \langle a_1, \ldots, a_n\rangle, a)$ transforms A into a new algebra B in the following way: $f^B(a_1, \ldots, a_n) = a$ and $f^B(\overline{a}) = f^A(\overline{a})$ for any tuple $\overline{a}$ different from $\langle a_1, \ldots, a_n\rangle$.[6]

Thus, the modifier $\mu 1$ either redefines a function at a certain point or defines it at a certain point if it has not yet been defined at this point. For example, $\mu 1$ applied to $\alpha = (matrix, <1,1>, 2)$ produces $matrix(1,1) = 2$.

Definition. A function update β in A is a pair $(f, \langle a_1, \ldots, a_n\rangle)$, where $a_i \in A_{T_i}, i = 1, \ldots, n$. The modifier $\mu 2$ applied in A to a $\beta = (f, \langle a_1, \ldots, a_n\rangle)$ transforms A into a new algebra B in the following way: f^B is undefined for $\langle a_1, \ldots, a_n\rangle$ and $f^B(\overline{a}) = f^A(\overline{a})$ for any tuple $\overline{a}$ different from $\langle a_1, \ldots, a_n\rangle$.

Thus, the modifier $\mu 2$ undefines a function at a certain point if it has been defined at this point. For example, $\mu 2$ applied to $\beta = (matrix, <1,1>)$ makes $matrix(1,1)$ undefined.

5.4. Creation of objects

The modifiers $\mu 1$ and $\mu 2$ can update the states of existing objects, they do not contribute to the creation of new object identifiers. A special algebra update serves for creating a new object identifier.[7]

Definition. Let T be an object type name. An update δ in A is a pair $\langle Var(T), id\rangle$ ($\langle Const(T), id\rangle$), where $id \in OID$. The modifier $\mu 3$ applied in A to δ transforms A into a new algebra B so that $B_{Var(T')} = A_{Var(T')} \cup \{id\}$ ($B_{Const(T')} = A_{Const(T')} \cup \{id\}$) for all T' such that $T \leq T'$.

Thus, the modifier $\mu 3$ expands the set of mutable (constant) object identifiers of a certain object type and all its supertypes. For example, if id is an object identifier, then $\mu 3$ applied to $\delta = (Var(Colorpoint), id)$ will change the current algebra A by inserting id into the sets $A_{Var(Colorpoint)}$ and $A_{Var(Point)}$.

[6]A strong equality is meant here and in the sequel.

[7]We do not consider the problem of object deletion since it more concerns the memory use optimization, which is an issue of a programming language rather than an issue of a specification language.

5.5. Update set

Definition. Let γ be a set of function updates. The set γ is *consistent* if it does not contain any two contradictory function updates of the following kind: $\alpha 1 = (f, \langle a_1, \ldots, a_n \rangle, a)$, $\alpha 2 = (f, \langle a_1, \ldots, a_n \rangle, a')$, and $\beta = (f, \langle a_1, \ldots, a_n \rangle)$, where $a \neq a'$ (two contradictory function updates define the function differently at the same point). The mutator μ applied in A to a consistent γ transforms A into a new algebra B by the simultaneous application of $\mu 1$ to all $\alpha \in \gamma$, $\mu 2$ to all $\beta \in \gamma$, and $\mu 3$ to all $\delta \in \gamma$. If γ is inconsistent, the new algebra in not defined. If γ is empty, B is the same as A. For example, μ applied to an update set

$\{(matrix, <1,1>, 2), (matrix, <2,2>), (Var(Colorpoint), id)\}$

will force $matrix(1,1)$ to produce 2, $matrix(2,2)$ to be undefined, and the sets of identifiers of $Var(Colorpoint)$ and $Var(Point)$ to contain id. At the same time, the result of applying μ to an update set

$\{(matrix, <1,1>, 2), (matrix, <1,1>), (Var(Person), id)\}$

is not defined.

The operation $\sqcup$ performs sequential union of two update sets. Let γ_1 and γ_2 be two consistent update sets, $update_1$ be one of the following updates: $\alpha_1 = (f, \langle a_1, \ldots, a_n \rangle, a)$, $\beta_1 = (f, \langle a_1, \ldots, a_n \rangle)$, and let $update_2$ be one of the following updates: $\alpha_2 = (f, \langle a_1, \ldots, a_n \rangle, a')$, $\beta_2 = (f, \langle a_1, \ldots, a_n \rangle)$, where $a \neq a'$. Then $u \in \gamma_1 \sqcup \gamma_2$ iff $u \in \gamma_1$ or $u \in \gamma_2$ except the following cases: if $update_1 \in \gamma_1$ and $update_2 \in \gamma_2$, then $update_2 \in \gamma_1 \sqcup \gamma_2$.

Thus, in the sequential union of update sets, each next update of a function at a certain point waives each preceding update of the function at the same point. If there are sequential creations of object identifiers of the same type, the set of object identifiers of this type will be expanded accordingly.
Example:

$\{(matrix, <1,1>, 2), (matrix, <2,2>, 0), (Var(Person), id)\} \sqcup$
$\{(matrix, <1,1>, 3), (Var(Person), id1)\} =$
$\{(matrix, <1,1>, 3), (matrix, <2,2>, 0), (Var(Person), id),$
$(Var(Person), id1)\}$.

We denote by Γ the set of all update sets in $D(A')$. We also introduce a notion of pair, $<\gamma, a>$, where γ is an update set and a is an algebra element. The function fst applied to $<\gamma, a>$ produces γ, and the function snd applied to $<\gamma, a>$ produces a.

5.6. Observers, transformers and procedures

We can now give semantics of observer, transformer, and procedure names. Given an algebra $A \in |D(A')|$, we associate:

- with each observer signature $b : T_1, \ldots, T_n \longrightarrow T'$ in an object type signature marked with T, a partial map (called an *observer*), $b_T^{D(A')}$, associating an element $a' \in A_{T'}$ with each pair $< A, < id, a_1, \ldots, a_n >>$, where $A \in |D(A')|$, $id \in A_T$, and $a_i \in A_{T_i}, i = 1, \ldots, n$. We write $b_T^{D(A')}(A, < id, a_1, \ldots, a_n >)$ for the application of $b_T^{D(A')}$ to $< A, < id, a_1, \ldots, a_n >>$;
- with each transformer signature $m : T_1, \ldots, T_n \longrightarrow T'$ in an object type signature marked with T, a partial map (an *ititializer* or *mutator*), $m_T^{D(A')}$, associating an update set $\gamma \in \Gamma$ and an element $a' \in A_{T'}$ with each pair $< A, < id, a_1, \ldots, a_n >>$, where $A \in |D(A')|$, $id \in A_T$, and $a_i \in A_{T_i}, i = 1, \ldots, n$. Only an update set is associated with the pair when m has either no profile or the profile $T_1, \ldots, T_n$. We write $m_T^{D(A')}(A, < id, a_1, \ldots, a_n >)$ for the application of $m_T^{D(A')}$ to $< A, < id, a_1, \ldots, a_n >>$.

Note that if $T < T'$ and both T and T' have an observer or transformer with the name f, different maps $f_T^{D(A')}$ and $f_{T'}^{D(A')}$ can be built in $D(A')$. This corresponds to the principles of *overloading* and *overriding* typical of object-oriented programming paradigm.

It is defined that a transformer produces a set of (function) updates serving for changing the state of an object with the identifier *id* (i.e., changing at least one attribute of *id*), which gives us a possibility to compare the updates made by two transformers (we would not be able to do this if transformers produced new algebras as dynamic operations do in [12]). Note that although a transformer can in principle be applied to a constant object, this facility is practically used only for the initialization of constant objects since one cannot construct a term indicating an update of a constant object (see next section). Note also that one can define a transformer or procedure changing the state and producing a value. Thus, an operation like *pop* popping a stack and producing its top element can be defined.

6. Terms and their interpretations

There are several rules for creating terms of object types. Moreover, a special kind of term called *transition term* is introduced to denote transitions from one algebra to another. Interpretation of these terms is done with the use of update sets. A mechanism of *message passing* is provided in this way. Let $D\Sigma = < \Sigma', OTYPE, O\Phi, int^o, DF >$ be a dynamic system signature, $D(A')$ be a dynamic system, a $D\Sigma$-algebra $A \in |D(A')|$ be an extension of a Σ'-algebra A', X be a set of $TYPE$-sorted variables and Y be a set of $OTYPE^o$-sorted variables (we denote the T-subset of the set by Y_T). Then we define the set of $D\Sigma$-terms, $T(D\Sigma, X, Y)$, as an extension of the set of

Σ'-terms, $T(\Sigma', X)$. Given a valuation function $u : Y \longrightarrow A$, we also define the interpretation t^A of a term $t \in T(D\Sigma, X, Y)$.

1. If $t \in T(\Sigma', X)$, then $t^A = t^{A'}$. A term of a many-sorted signature is interpreted conventionally.
2. If $y \in Y_T$, then y is a term of type T. **Interpretation**: $y^A = u(y)$.
3. If $T \in OTYPE$, then both a term of type $Var(T)$ and a term of type $Const(T)$ are terms of type T. **Interpretation**: if $(t : Var(T))^A$ evaluates to an $id \in A_{Var(T)}$ or $(t : Const(T))^A$ evaluates to an $id \in A_{Const(T)}$, then the same id is the interpretation of $(t : T)$.
4. If $T_1 \leq T$ and t is a term of type T_1, then t and $t(T)$ are terms of type T, too. **Interpretation**: if t evaluates in A to an $id \in A_{T_1}$, then by definition id also belongs to A_T and can be used as an object identifier of type T. The first form of the term is used in the context where a term of type T is needed, while the second form is used when t serves for the invocation of a mutator defined in the type T[8].

   If there is no T_2 such that $T_2 < T_1$ and $t^A \in A_{T_2}$, then the type T_1 is called the *static type* of t and T is called the *dynamic type* of t.
5. If t is a term of type T and $T_1 < T$, then $t(T_1)$ is a term of type T_1. **Interpretation**: $t(T_1)^A = t^A$ if $t^A \in A_{T_1}$; $t(T_1)^A$ is undefined otherwise. This is a *typeguard* like that one in Oberon [17].
6. If t is a term of type T, then $\mathbf{D}(t)$ is a term of type Boolean. **Interpretation**: $\mathbf{D}(t)^A = true^A$ if t is defined in A, and $\mathbf{D}(t)^A = false^A$ otherwise.
7. If $at : T_1, \ldots, T_n \longrightarrow T'$ is an attribute signature from the object type signature marked with T, $t_1, \ldots, t_n$ are terms of types $T_1, \ldots, T_n$, respectively, and t is a term of type T, then $t.at(t_1, \ldots, t_n)$ is a term of type T' called an *attribute value*. **Interpretation**:
   $$t.at^A = at_T^A(t^A)(t_1^A, \ldots, t_n^A)$$
   if t and each t_i, $i = 1, \ldots, n$, are defined in A and at_T^A is defined for $< t^A, t_1^A, \ldots, t_n^A >$; $t.at(t_1, \ldots, t_n)^A$ is undefined otherwise. Thus, an attribute value is produced by the corresponding attribute function.
8. If $b : T_1, \ldots, T_n \longrightarrow T'$ is an observer signature from the object type signature marked with T, $t_1, \ldots, t_n$ are terms of types $T_1, \ldots, T_n$, respectively, and t is a term of type T, then $t.b(t_1, \ldots, t_n)$ is a term of type T' called an *observer call*. **Interpretation**: if T_1 is the static type of t, then
   $$t.b(t_1, \ldots, t_n)^A = b_{T_1}^{D(A')}(A, < t^A, t_1^A, \ldots, t_n^A >)$$

[8]The last form generalizes the construction **super** often used in object-oriented PLs.

if t and each $t_i, i = 1, \ldots, n$, are defined in A and $b_{T_1}^{D(A')}$ is defined for $< A, < t^A, t_1^A, \ldots, t_n^A >>$; $t.b(t_1, \ldots, t_n)^A$ is undefined otherwise.

Thus, the interpretation of an observer call leads to the invocation of the observer associated with b in the static type of the term t with the corresponding object identifier as argument and with some other arguments if any. In this way, dynamic (late) binding of an observer name with the corresponding observer is provided.

9. If $m : T_1, \ldots, T_n \longrightarrow T'$ is a transformer signature from the object type signature marked with T, $t_1, \ldots, t_n$ are terms of types $T_1, \ldots, T_n$, respectively, and t is a term of type $Var(T)$, then $t.m(t_1, \ldots, t_n)$ is a transition term of type T' (a transition term of type **void** if m has profile $T_1, \ldots, T_n$) called a *transformer* (*initializer, mutator*) *call.* This kind of term serves for indicating an update of a mutable object (one cannot update a constant object).

Interpretation 1. If m is an intializer name, then

- if T is the static type of t, then
$$t.m(t_1, \ldots, t_n)^A = m_T^{D(A')}(A, < t^A, t_1^A, \ldots, t_n^A >)$$
if t and each $t_i, i = 1, \ldots, n$, are defined in A and $m_T^{D(A')}$ is defined for $< A, < t^A, t_1^A, \ldots, t_n^A >>$. Otherwise $t.m(t_1, \ldots, t_n)^A$ is undefined.
- if the static type of t is different from T, then $t.m(t_1, \ldots, t_n)^A$ is undefined.

Thus, the interpretation of an initializer call leads to the invocation of the indicated initializer with the corresponding object identifier as argument and with some other arguments if any. Note that one cannot initialize an object if its static and dynamic types are different.

Interpretation 2. If m is a mutator name and T_1 is the static type of t, then
$$t.m(t_1, \ldots, t_n)^A = m_{T_1}^{D(A')}(A, < t^A, t_1^A, \ldots, t_n^A >)$$
if t and each $t_i, i = 1, \ldots, n$, are defined in A and $m_{T_1}^{D(A')}$ is defined for $< A, < t^A, t_1^A, \ldots, t_n^A >>$. Otherwise $t.m(t_1, \ldots, t_n)^A$ is undefined.
Thus, the interpretation of a mutator call leads to the invocation of the mutator associated with m in the static type of the term t with the corresponding object identifier as argument and with some other arguments if any. In this way, dynamic (late) binding of a mutator name with the corresponding mutator is provided.

10. If T is an object type name, then $new\_var(T)$ is a transition term of type T. **Interpretation**:

$$new\_var(T)^A = < \{\delta\}, id >,$$

where $id \in OID$, $\delta = < T, id >$ and $id \notin A_T$ for any $T \in OTYPE$. Thus, the interpretation of this transition term leads to the creation of a new mutable object identifier with totally undefined attribute functions. Note that the interpretations of several terms $new\_var(T)^A$ in the same instance algebra A always produce the same object identifier id.

11. If T is an object type name, $m : T_1, \ldots, T_n$ is a transformer signature from the object type signature marked with T, and $e_1, \ldots, e_n$ are terms of types $T_1, \ldots, T_n$, respectively, then $new\_const(T'm(e_1, \ldots, e_n))$ is a transition term of type T. **Interpretation**. Let B be an algebra produced by $\mu 3(A, \delta)$, where $\delta = < Const(T), id >$, where $id \in OID$ and $id \notin A_T$ for any $T \in OTYPE$. Then

$$new\_const(T'm(e_1, \ldots, e_n))^A = < \{\delta\} \cup \gamma)), id >,$$

where γ is an update set such that $\mu(B, \gamma) = m_T^{D(A')}(B, id, e_1^A, \ldots, e_n^A)$. If at least one of $e_1, \ldots, e_n$ is not defined in A, the result is undefined.

Thus, the interpretation of this transition term leads to the creation of a new constant object identifier provided with attributes defined by the transformer call indicated. Note that the interpretations of several terms $new\_const(T'm(\ldots))^A$ in the same instance algebra A always produce the same object identifier id, which leads to an error if the initializations are different.

7. Transition rules

Algebra updates are specified by means of special *transition rules*[9]. A transition rule is a special kind of transition term. It is applicable only to a *dynamic* function (constant) which can evolve from one algebra to another. In addition to the dynamic functions (constants) introduced in Section 5.2, an attribute function of a mutable object is also a dynamic function (constant). This means that an attribute at defined in an object type T is a dynamic function only if it is applied to a term t of type $Var(T)$ (t cannot denote a constant object). We usually write "transition rule" (or simply "rule") for "transition term of type **void**". The semantics of transition terms is defined in terms of update sets (and values) produced.

7.1. Basic transition rules

Definition. Let $f : T_1, \ldots, T_n \longrightarrow T$ be a dynamic function signature, t_i, $i = 1, \ldots, n$, be a ground term of type T_i and t be either a ground term of

[9]The set of transition rules proposed in the paper is based on the set of basic rules of [9] and affected by [8].

type T or a transition term of type T. Then

$f(t_1, ..., t_n) := t$ and $f(t_1, ..., t_n) := undef$

are transition rules called *primitive update instructions.*

Interpretation 1. If A is an instance $D\Sigma$-algebra, $t_i, i = 1, \ldots, n$, is defined in A and t is a term of type T, then

$(f(t_1, ..., t_n) := t)^A = \{\alpha\}$ if t is defined in A,

$(f(t_1, ..., t_n) := t)^A = \{\beta\}$ if t in not defined in A,

$(f(t_1, ..., t_n) := undef)^A = \{\beta\}$, where

$\alpha = (f, < t_1^A, \ldots, t_n^A >, t^A)$ and $\beta = (f, < t_1^A, \ldots, t_n^A >)$.

If at least one of $t_i, i = 1, \ldots, n$, is not defined in A, then both $(f(t_1, ..., t_n) := t)^A$ and $(f(t_1, ..., t_n) := undef)^A$ are undefined.

Examples. Let x be a dynamic constant of type Nat and f be a dynamic function from Nat to Nat. The execution of the transition rule

$f(x) := f(x) + 1$

will transform an algebra A into an algebra B so that $f^B(x^A) = f^A(x^A) + 1$. A transition rule

$x := undef$

will make x undefined in the new algebra.

If $at : T'$ is an attribute signature from an object type T, o is a ground term of type $Var(T)$ and t is a ground term of type T', then the transition rules

$o.at := t$ and $o.at := undef$

are interpreted as $at(o) := t$ and $at(o) := undef$.

Examples. Let c be a term denoting a mutable object of type *Colorpoint* and v be a term denoting a mutable object of type *Person*. Then the transition rule

$c.x := 1$

will transform an algebra A into an algebra B so that

$x^B_{Colorpoint}(c^A) = 1$ and $x^B_{Point}(c^A) = 1$

(recall that the same function is associated with both attribute names). A transition rule

$v.spouse := undef$

will change the algebra in such a way that $spouse^B_{Person}(v^A)$ is not defined.

If we have two dynamic constants, say $o1$ and $o2$, of the same object type T, then the transition rule:

$o1 := o2$

will force both of them to have the same object identifier, which provides for *object sharing.*

Interpretation 2. If t is a transition term of type T, then

$(f(t_1, ..., t_n) := t)^A = \gamma_1 \sqcup \gamma_2$,
where $\gamma_1 = fst(t^A)$ and $\gamma_2 = \{f, < t_1^A, \ldots, t_n^A >, snd(t^A)\}$ (both parts of an update instruction are evaluated in the same algebra); $(f(t_1, ..., t_n) := t)^A$ is undefined if at least one of $t, t_i, i = 1, \ldots, n$, is not defined in A.

Examples. Let *dates* and *last_holiday* be dynamic constants of types $Var(Date)$ and $Const(Date)$, respectively. Then the execution of the transition rule:

dates := new_var(Date)

will create a new mutable object of type *Date* and assigns it to *dates*, and the execution of the transition rule:

last_holiday := new_const(Date'create_date(1, 5, 1998))

will create a new constant object indicating a concrete date and assign it to *last_holiday*.

7.2. Rule constructors

Complex transition terms are constructed recursively from update instructions by means of several rule constructors, e.g., the *sequence constructor*, *set constructor*, the *condition constructor*, *guarded update*, etc. We define here only the *sequence constructor*, *set constructor*, *guarded update*, and *loop constructors* because the others have minimal relevance to the subject of the paper (one can find more details in [1, 11, 10]).

Sequence constructor. If $R_1, R_2, \ldots, R_n$ are transition rules and t is a term of type T, then **seq** $R_1, R_2, \ldots, R_n$ **res** t **end** is a transition term of type T and **seq** $R_1, R_2, \ldots, R_n$ **end** is a transition term of type **void**.

Interpretation. Let $A_1, A_2, \ldots, A_n$ be algebras, such that $A_1 = R_1^A$, $A_2 = R_2^{A_1}, \ldots, A_n = R_n^{An-1}$, and let $\gamma_1, \gamma_2, \ldots, \gamma_n$ be update sets, such that $A_1 = \mu(A, \gamma 1)$, $A_2 = \mu(A_1, \gamma 2)$, $\ldots$, $A_n = \mu(A_{n-1}, \gamma_n)$. Then

seq $R_1, R_2, \ldots, R_n$ **res** t $\mathbf{end}^A = < \gamma, t^{A_n} >$,

seq $R_1, R_2, \ldots, R_n$ $\mathbf{end}^A = \gamma$,

where $\gamma = \gamma_1 \sqcup \gamma_2 \sqcup \ldots \sqcup \gamma_n$.

Thus, to execute a sequence of rules starting with an algebra A, it is sufficient to create sequential union of their update sets and use it for the transformation of A (which is equivalent to the sequential execution of the rules one after another). If the rule contains a resulting expression, it is evaluated in the resulting algebra.

The set constructor. If $R_1, \ldots, R_n$ are transition rules and t is a term of type T, then **set** $R_1, \ldots, R_n$ **res** t **end** is a transition term of type T and **set** $R_1, \ldots, R_n$ **end** is a transition term of type **void**.

Interpretation. Let $A_1 = R_1^A, \ldots, A_n = R_n^A$ and let $\gamma_1, \ldots, \gamma_n$ be update sets, such that $A_1 = \mu(A, \gamma 1), \ldots, A_n = \mu(A, \gamma_n)$. Then

set $R_1, \ldots, R_n$ **res** t **end**$^A = < \gamma, t^A >$,

set $R_1, \ldots, R_n$ **end**$^A = \gamma$,

where $\gamma = \gamma_1 \cup \ldots \cup \gamma_n$.

In other words, to execute a set of rules, execute all of them in parallel and unite the results. If the rule contains a resulting expression, it is evaluated in the source algebra.

Example: Let x, y, z be dynamic constants of type *Nat* and f be a dynamic function *Nat* to *Nat*. Then the execution of a set of rules:

set $f(x) := y, y := x, x := z$ **end**

will produce: $f^B(x^A) = y^A$; $y^B = x^A$; $x^B = z^A$.

A special notation is introduced for the parallel update of all attributes of a mutable object. Let T be an object type with attribute names $at_1, \ldots, at_k$, o be a term of type $Var(T)$ and $o1$ be a term of type T. Then a transition rule:

$o \uparrow := o1 \uparrow$

is equivalent to:

set $o.at_1 := o1.at_1, \ldots, o.at_k := o1.at_k$ **end**.

A **guarded update** instruction is a rule of the form **if** g **then** R, where R is a rule.

Interpretation. Let $R1 =$ **if** g **then** R. Then $R1^A = R^A$ if $g^A = true^A$; $R1^A = \oslash$ otherwise. In other words, execute the rule if the condition evaluates to *true* and do nothing in the opposite case.

Loop constructors. The guarded update together with the sequence constructor gives us a possibility to define some loop constructors. If R is a rule and g is a Boolean term, then

while g **do** R and

do R **until** g

are transition rules.

Interpretation.

(**while** g **do** $R)^A =$ (**if** g **then seq** R, **while** g **do** R **end**)^A;

(**do** R **until** $g)^A =$ (**seq** R, **if**$\neg g$ **then do** R **until** $g)^A$.

7.3. Massive update

A massive update permits the specification of a parallel update of one or more functions at several points. It has the following form:

forall $x_1 : T_1, \ldots, x_n : T_n . R$,

where $x_1, \ldots, x_n$ are bound variables of types $T_1, \ldots, T_n$, respectively, and R

is a transition rule having no free variables.
Interpretation. For all $t_1 \in T(D\Sigma)_{T_1}, ..., t_n \in T(D\Sigma)_{T_n}$, where $T(D\Sigma)_{T_i}$ is the set of ground $D\Sigma$-terms of type T_i,

$$(\textbf{forall } x_1 : T_1, ..., x_n : T_n.R)^A = \bigcup\{(R[t_1/x_1, ..., t_n/x_n])^A\}.$$

Example. Let f be a dynamic function from Nat to Nat. A transition rule

forall x: Nat. $f(x) := f(x) + 1$

is equivalent to the set of rules

$\{(f(t) := f(t) + 1) : \ t \in T(D\Sigma)_{Nat}\}$.

This means that $f^B(t^A) = f^A(t^A) + 1$ for all t such that $f^A(t^A)$ is defined.

The massive update allows us to interpret copying of complex attributes. Thus if $at : T_1, \ldots, T_n \longrightarrow T'$, where n is greater than zero, is an attribute signature in an object type T and $o, o1$ are two objects of type $Var(T)$, then the update instruction $o.at := o1.at$ is interpreted as follows:

forall $x_1 : T_1, \ldots, x_n : T_n.\ o.at(x_1, \ldots, x_n) := o1.at(x_1, \ldots, x_n)$.

8. Dynamic system specification

If t_1 and t_2 are two $D\Sigma$-terms of type T, then $t_1 == t_2$ is a *static equation*. Let SE be a set of static equations. An instance algebra A is a model of SE if it satisfies each equation in SE. Let $|D(A')|$ be the carrier of a dynamic system $D(A')$, such that each algebra $A \in |D(A')|$ is a model of SE.

If t_1 and t_2 are transition terms, then $t_1 == t_2$ is a *dynamic equation*. If t_1 and t_2 are transition terms of type T, then a dynamic equation holds in $D(A')$ iff for any algebra $A \in |D(A')|$ there is an update set γ and a value a of type T such that $t_1^A = <\gamma, a>$ and $t_2^A = <\gamma, a>$. If t_1 and t_2 are just transition rules, then a dynamic equation holds in $D(A')$ iff for any algebra $A \in |D(A')|$ there is an update set γ such that $t_1^A = \gamma$ and $t_2^A = \gamma$. This means that the transformation of A according to either t_1 or t_2 should produce the same algebra B.

If a dynamic equation de holds in $D(A')$, we say that $D(A')$ is a *model* of de. A dynamic equation de is *consistent* if there is at least one model of it. Example: if $c1$ and $c2$ are constants of two different types T_1 and T_2, respectively, and t_1 and t_2 are terms of types T_1 and T_2, respectively, then an equation

$c1 := t_1 == c2 := t_2$

is inconsistent since in $D(A')$ the updates $(c1, t_1)$ and $(c2, t_2)$ are different.

If DE is a set of consistent equations, then $D(A')$ is a model of DE if each $de \in DE$ holds in $D(A')$.

Definition. Let $D\Sigma = <\Sigma', OTYPE, O\Phi, int^o, DF>$ be an object-structured signature, $<\Sigma', SE'>$ be the specification of its static part, and each $o\phi \in O\Phi$ be accompanied with a set of static equations for each observer name in $o\phi$, a set of dynamic equations for each transformer name in $o\phi$, and a set of dynamic equations for each procedure name in DF. Then we get a *dynamic system specification*, $< D\Sigma, E >$. A pair $< o\phi, E^{o\phi} >$, where $E^{o\phi}$ is a set of equations associated with observer and transformer names from $o\phi$, is called an *object type specification.* A pair $< ps, E^{p} >$, where ps is a procedure signature from DF and E^{ps} is a set of equations associated with this procedure signature, is called a *procedure specification.*

Notation. In the following specifications, the set of equations is enclosed in curly brackets; the clause **dom** specifies the domain of a partial function: in a domain specification **dom** $t : b$, t is defined if and only if b evaluates to *true.*

Examples.

```
class SeqOfNat = spec
[mutator  empty; – construction of an empty sequence
    append: Nat; – appending a natural number to a sequence
    delete_head: ⟶ Nat; – deleting the head of a sequence
 attribute head: Nat; – fetching the head of a sequence
    tail: SeqOfNat; – fetching the tail of a sequence
    is_empty: Boolean; – checking whether a sequence is empty
 observer has: Nat ⟶ Boolean; – checking for the presence of an element]
{forall s, temp: Var(SeqOfNat), x: Nat. – set of universally quantified variables
 dom s.delete_head: ¬s.is_empty;
                       – one cannot delete the head of an empty sequence
s.empty == set s.head := undef, s.tail := undef, s.is_empty := true end;
s.append(x) ==
    seq temp := new_var(SeqOfNat), – a new identifier for the tail is allocated
        set s.head := x, – "x" is now the head of the sequence
            s.tail := temp,
            s.is_empty := false,
            temp↑ := s↑ – attributes of "s" are assigned to its tail
        end
    end;
s.delete_head == set s↑ := s.tail↑ res s.head end;
                                   – "s" gets the attributes of its tail
s.has(x) == ¬ s.empty & (s.head = x | s.tail.has(x))};

class Date = spec
[initializer create_date: Nat, Nat, Nat; – creating a date
 attribute day, month, year: Nat – the value of a day, month or year]
{forall date: Var(Date), d, m, y: Nat.
 date.create_date(d, m, y) ==
        set date.day := d, date.month := m, date.year := y end};

class Person = spec
[initializer create_person: String, Const(Date);
 mutator marry: Person;
          divorce;
 attribute name: String;
          spouse: Person;
          birth_date: Const(Date)]
{forall p, p1: Var(Person), n: String, d: Const(Date).
 dom p.marry(p1): ¬D(p.spouse) & ¬D(p1.spouse); - one cannot marry twice
 p.create_person(n, d) ==
     set p.name := n, p.birth_date := d, p.spouse := undef end;
 p.marry(p1) == set p.spouse := p1, p1.spouse := p end;
 p.divorce ==
     if D(p.spouse) then set p.spouse := undef, p.spouse.spouse := undef end};

class Rectangle = spec
[initializer create: Nat, Nat; - creating a new rectangle
          default_rectangle; - default rectangle
 attribute length, width: Nat - getting rectangle parameters;
 observer area: Nat - computing a rectangle's area;
          equal: Rectangle ⟶ Boolean; - comparison of rectangles for equality]
{forall r: Var(Rectangle), r1, r2: Rectangle, x, y: Nat.
 r.default_ rectangle == set r.length := 0, r.width := 0 end;
 r.create(x, y) == set r.length := x, r.width := y end;
 r1.area == r1.length * r1.width;
 r1.equal(r2) == r1.length = r2.length & r1.width = r2.width}.
```

9. Object type specification methodology

To provide a variety of ways an object can be manipulated, we distinguish between *directly updatable* attributes and *indirectly updatable* attributes. A directly updatable attribute can be updated by a transition rule. For example, in the object type Rectangle specified above, the attributes *length* and *width* of an object s can be directly updated by transition rules like $s.length := x$ and $s.width := y$. An indirectly updatable attribute can be updated only by a transformer. For example, in the object type $SeqOfNat$ specified above, the attribute *head* can be updated by mutators *append* and *delete_head*.

Notation: we will introduce directly updatable attributes with the keyword **direct** and indirectly updatable attributes with the keyword **indirect**.

Examples. Two of the previous four examples can be now rewritten in the following way:

```
class SeqOfNat = spec
 [mutator empty; – construction of an empty sequence
      append: Nat; – appending a natural number to a sequence
      delete_head: ⟶ Nat; – deleting the head of a sequence
  indirect head: Nat; – fetching the head of a sequence
      tail: SeqOfNat; – fetching the tail of a sequence
      is_empty: Boolean; – checking whether a sequence is empty
  observer has: Nat ⟶ Boolean; – checking for the presence of an element]
 {forall s, temp: Var(SeqOfNat), x: Nat. – set of universally quantified variables
  dom s.delete_head: ¬s.is_empty;
                           – one cannot delete the head of an empty sequence
  s.empty == set s.head := undef, s.tail := undef, s.is_empty := true end;
  s.append(x) ==
      seq temp := new_var(SeqOfNat), – a new identifier for the tail is allocated
           set s.head := x, – "x" is now the head of the sequence
                 s.tail := temp,
                 s.is_empty := false,
                 temp↑ := s↑ – attributes of "s" are assigned to its tail
           end
      end;
  s.delete_head == set s↑ := s.tail↑ res s.head end;
                                          – "s" gets the attributes of its tail
  s.has(x) == ¬ s.empty & (s.head = x | s.tail.has(x))};

class Rectangle = spec
 [mutator default_rectangle; – default rectangle
       create: Nat, Nat; – creating a new rectangle
  direct length, width: Nat – getting rectangle parameters;
  observer area: Nat – computing a rectangle's area;
       equal: Rectangle ⟶ Boolean; – comparison of rectangles for equality]
 {forall r: Var(Rectangle), r1, r2: Rectangle, x, y: Nat.
  r.default_ rectangle == set r.length := 0, r.width := 0 end;
  r.create(x, y) == set r.length := x, r.width := y end;
  r1.area == r1.length * r1.width;
  r1.equal(r2) == r1.length = r2.length & r1.width = r2.width}.
```

●

Definition. An object type is properly specified if any change of its state produced by a transformer unambiguously defines the values of its attributes.

Fact 1. An object type is properly specified if:

1) for each transformer, there is an equation relating a transformer call to the values of the direct and indirect attributes of the caller (by definition, an attribute value remains the same if a new value is not assigned to it);
2) for at least one transformer, there is an equation relating a mutator call to the values of all the direct and indirect attributes of the caller (an object is fully initialized);
3) for each observer, there is an equation defining an observer call in terms of values of direct and/or indirect attributes of the caller and/or arguments if any.

The proof of the fact is self-evident.

Fact 2. A properly specified object type has a model.
Following Fact 1, the construction of such a model is a trivial task.

Here is an example of an object type specification with partially updated attributes:

```
class Circle = spec
[initializer create: Real, Real, Real, Color;
 mutator move: Real, Real;
         resize: Real;
         changeCol: Color;
 indirect X, Y, radius: Real; – indirect attributes
         col: Color – another indirect attribute]
{forall c: Var(Circle), x, y, r: Real, cl: Color.
 c.create(x, y, r, cl) ==
         set c.X := x, c.Y := y, c.radius := r, c.col := cl end;
 c.move(x, y) == set c.X := c.X + x, c.Y := c.Y + y end;
         – radius and color do not change
 c.resize(r) == c.radius := c.radius * r; – X, Y, and color do not change
 c.changeCol(cl) == c.col := cl; – X, Y, and radius do not change}.
```

Note that the initializer *create* fully defines an object's state, whereas any mutator partially updates the state. However, each object state can be put in one-to-one correspondence with a call of *create*.

10. Related work

Although the idea of modeling states as algebras is not very new and it has been elaborated in some works in programming language semantics [19, 20, 21], the most related work is the evolving algebra approach of Gurevich [9], algebraic specifications with implicit state of Dauchy and Gaudel [8], d-oids of Astesiano and Zucca [12], and dynamic abstract types of Ehrig and Orejas [13].

The evolving algebras approach has provided a mechanism for the transition from one state to another by means of transition rules resembling imperative programming statements. As a result, a specification looks like an imperative program, it is easier to understand and is executable. The present work uses Gurevich's transition rules as a means of object type specification.

The idea to represent states of a system by algebras and dynamic operations by transformations between them is also advocated by Dauchy and Gaudel [8, 11]. Transformations are defined by means of so-called modifiers which are counterparts of transition rules of Gurevich Machines. The main contribution of the work is the proposition of a specification mechanism for modifiers. This idea is used in the present work as the basis of object mutator specifications.

The same idea of an implicit state in terms of a new mathematical structure, d-oid (dynamic object identity), is given by Astesiano and Zucca [12]. A d-oid is a set of instance structures (e.g., algebras) and a set of dynamic operations (transformations of instance structures with a possible result of a definite sort). Here dynamic operations serve as counterparts of transition rules of Gurevich and modifiers of Dauchy and Gaudel. Algebra elements are considered as objects supplied with unique identifiers preserving object identities in the process of algebra transformations. 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 aim.

Dynamic abstract types are informally introduced in [13]. 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, instance structure specification, dynamic operation specification, and higher-level specification. In contrast to this approach, our aim is to separate strictly conventional data types and dynamic object types, providing each of them with only one level of specification.

The idea of dynamic types is also investigated in [22]. 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. The present paper proposes in addition both a formal definition of a dynamic object type and an approach to its formal specification.

A simplified approach taking into account only mutable objects without subtyping is proposed in [23]. These shortcomings are eliminated in the present paper.

An extension of a temporal logic for specifying and reasoning about object classes and their instances is presented in [24]. We are still satisfied with a conventional many-sorted logic and have shown its applicability to dynamic object specifications.

Finally, a fundamental work [27], where an object is just a tuple of methods (no notion of object identifier) and object updates are simulated by method overrides, should be mentioned. In contrast to this work, we prefer to have objects supplied with unique identifiers and to have permanent methods for all objects of the same type, which better corresponds to the conventional paradigm of object-orientedness and produces a specification much better understood by a programmer.

11. Conclusion and further work

The main contribution of this work is the use of transition rules of an Abstract State Machine (former known as Evolving Algebra) for the specification of object types. For this purpose, an original algebraic model of an object type has been elaborated. It has two outstanding features: it naturally models the notions of object and object class in modern programming languages, and it allows an object type to be specified in the manner resembling the specification of conventional data types. Each object possesses a state and a unique identifier. For each object type, a set of unique identifiers is provided by an algebra of the corresponding signature. An object state is represented by a number of updatable observers defined as functions from object identifiers to values of some other (attribute) types. An object type signature introduces the observers of the corresponding set of objects and mutators serving to initialize and update observers of particular objects. An advantage of the technique proposed is that an object specification is done in an abstract and precise way, the specification is executable, and it is easy to understand by programmers.

We create only an identifier sort for each object type in contrast to [25, 26] where value sorts are also created. This significantly simplifies the semantics of an object update and permits us to specify objects types with mutual references. Constant objects in addition to mutable objects are allowed. This facility helps us to model immutable objects admissible in some programming languages (e.g., C^{++}) and some data models.

A dynamic system is represented as a set of object-structured instance algebras with operations transforming one algebra into another. The specification of such a system consists of specifications of data types representing the static part of the system, object types representing the dynamic part of the system, and independent constants, functions and mutators corresponding to variable, functions and procedures of conventional imperative languages.

We did not touch in this paper on such an important issue as generic object types. This remains a subject of further research.

References

[1] A.V. Zamulin, *Typed Gurevich Machines Revisited*, Joint NCC&ISS Bull., Comp. Science, 7, 1997, 93–121 (available electronically from http://www.eecs.umich.edu/gasm/).

[2] P. Wegner, *Dimensions of object-oriented language design*, ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications, 1987, 168–182.

[3] B.C. Pierce and D.N. Turner, *Simple Type-Theoretic Foundation For Object-Oriented Programming*, J. Functional Programming 1 (1): 1-000, January 1993.

[4] C. Beeri, *A Formal approach to object-oriented Databases*, Data&Knowledge Engineering (5), 1990, 353–382.

[5] J.A. Goguen and R. Diaconescu, *Towards an algebraic semantics for the object paradigm*, Recent Trends in Data Type Specification, LNCS, 1994, **785**, 1–29.

[6] F. Parisi-Presicce and A. Pierantonio, *Structured inheritance for algebraic class specifications*, Recent Trends in Data Type Specification, LNCS, 1994, **785**, 295–309.

[7] A.V. Zamulin, *Algebraic specification of an Oberon Target Machine*, Proc. A.P. Ershov Second Intern. Memorial Conference "Perspectives of System Informatics", Novosibirsk, June 25–28, 1996, LNCS, **1181**, 41–54.

[8] P. Dauchy and M.C. Gaudel, *Algebraic Specifications with Implicit State*, Tech. report, No. 887, Iniv. Paris–Sud, 1994.

[9] Y. Gurevich, *Evolving Algebras 1993: Lipary Guide*, Specification and Validation Methods, Oxford University Press, 1994.

[10] Y. Gurevich, *May 1997 Draft of the ASM Guide*. Available electronically from http://www.eecs.umich.edu/gasm/.

[11] C. Khoury, M.C. Gaudel and P. Dauchy, *AS–IS*, Tech. report, No. 1119, Iniv. Paris–Sud, 1997.

[12] E. Astesiano and E. Zucca, *D-oids: a model for dynamic data types*, Mathematical Structures in Computer Science, 5(2), June 1995, 257–282.

[13] H. Ehrig and F. Orejas, *Dynamic abstract types: an informal proposal*, Bull of EATCS, 53, June 1994, 162–169.

[14] M. Wirsing, *Algebraic Specifications*, Handbook of Theoretical Computer Science, Elsevier Science Publishers B.V., 1990, 665–788.

[15] A.V. Zamulin, *The database specification language RUSLAN: main features*, East-West database Workshop (proc. Second International East-West Database Workshop, Klagenfurt, Austria, September 25–28, 1994), Springer (Workshops in Computing), 1994, 315–327.

[16] G. Smalka, W. Nutt, J. A. Goguen and J. Meseguer. *Order-Sorted Equational Computation.* In: H. Ait-Kaci and M. Niva, eds., Resolution of Equations in Algebraic Structures, vol. 2, Academic Press, New York, 1989, pp. 299-367.

[17] N. Wirth. *The Programming Language Oberon (Revised edition).* Department Informatik, Institut fur Computersysteme, ETH, Zurich, 1990.

[18] R. Groenboom and R. Renardel de Lavalette, *Reasoning about dynamic features in specification languages*, Workshop in Semantics of Specification Languages, Springer Verlag, 1994, 340–355.

[19] H. Ganziger, *Denotational semantics for languages with modules*, D. Bjorner, editor, Formal Description of Programming Concepts II, North–Holland, 1983, 3–21.

[20] M.C. Gaudel, *Correctness proof of programming language translations*, D. Bjorner, editor, Formal Description of Programming Concepts II, North–Holland, 1983, 25–43.

[21] *Algebraic Methods: Theory, Tools and Applications*, M. Wirsing and J.A. Bergstra, editors, LNCS, No. 394, 1987.

[22] E. Zucca, *From static to dynamic data types*, W. Penchek and A. Szalas, editors, Mathematical Foundations of Computer Science 1996, LNCS, **1113**, 1996, 579–590.

[23] A.V. Zamulin, *Algebraic specification of dynamic objects*, Proc. Intern. Conf. "Languages et Models with Objets", Roscoff, Bretagne, France, October 22–24, 1997.

[24] A. Sernadas and C. Sernadas, *Object specification logic*, Journal of Logic and Computation, 5(5), 1995, 603–630.

[25] A. Pierantonio. *Making Statics Dynamic.* In: G. Hommel, editor, Proc. International Workshop on Communication based Systems, Kluwer Academic Publishers, 1995, pp. 19-34.

[26] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kush. *Revised Version of the Modelling Language TROLL.* Technishe Universitaet Braunschweig, Informatik-Berichte 94-03, 1994.

[27] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.