Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

A. V. Zamulin

ADDING GENERICITY TO OBJECT-ORIENTED ASMS

Preprint
60

Novosibirsk 1999

Facilities for the definition of generic object types, generic type categories, generic functions and generic procedures in an object-oriented ASM are described in the paper. These facilities permit one to specify algorithms over complex data structures abstracting both from the type of the structure components and the structure itself. The use of the facilities is demonstrated by the specifications of some important parts of Standard Template Library for C++.

Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

А. В. Замулин

РОДОВЫЕ СРЕДСТВА В ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ МАШИНАХ АБСТРАКТНЫХ СОСТОЯНИЙ

Препринт
60

Новосибирск 1999

Предлагаются средства формального определения родовых типов объектов, сатегорий родовых типов, годовых функций и родовых процедур в объектно-ориентированных машинах абстрактных состояний. Эти средства позволяют специфицировать алгоритмы над сложными структурами данных, абстрагируясь как от типа компонентов структуры, так и самой структуры. Использование предложенных средств демонстрируется примерами спецификаций некоторых важных компонентов библиотеки стандартных шаблонов для языка программирования C++.

1. INTRODUCTION

Object-oriented ASMs as a variant of traditional ASMs [1, 2] are formally introduced in [3]. They permit the specification of a dynamic system in terms of mutable and constant objects belonging to different object types. Unfortunately, the technique described in the paper does not provide facilities for the specification of generic object types and generic algorithms. At the same time, such facilities are highly needed if one wishes to write reusable specifications.

As a typical example, let us consider the problem of the specification of the Standard Template Library (STL) for C++ [4] which will be used in the paper as a running example. The semantics of the library components is currently given only informally as a set of requirements stated partly in English and partly as C++ program fragments. As a result, the semantics remains incomplete and imprecise, depending heavily on reader's (and library implementor's) intuition and knowledge of C++. Therefore, a formal description of STL independent of a particular programming language is highly needed.

STL is based on the notion of *container* which is a data structure consisting of a number of elements of the same type. Several container classes are defined in STL: vectors, lists, deques, sets, multisets, maps, and multimaps. Other container classes can be defined if needed. Each container class is parameterized by the component type. Thus, for each data structure one can write an algorithm abstracting from the component type. This provides the first level of genericity typical of C++.

To abstract from the container's structure, STL introduces the notion of *iterator* which is a generalization of the pointer notion. Iterators are grouped into different iterator categories providing abstract data-accessing methods. There are categories of input iterators, output iterators, forward iterators, bidirectional iterators, and random-access iterators. Iterator categories build a hierarchy. This means that each forward iterator is also an input iterator and an output iterator, each bidirectional iterator is also a forward iterator, and each random-access iterator is also a bidirectional iterator. Algorithms can now be written in terms of iterators abstracting from a concrete data structure. Most important here is that an algorithm requiring, say, an input iterator can also use a forward or bidirectional or random-access iterator. This provides the second level of genericity.

[0]The research is supported in part by Russian Foundation for Basic Research under the grant No 98-01-00682.

One of the ways of the formal STL definition is the use of classical algebraic specifications whose advantages are sound mathematical foundation and existence of specification languages and tools. Taking into account the generic nature of the data structures and iterator categories, a specification language like Spectrum [5] providing parameterized sorts and sort classes can be used for this purpose. However, the notions of iterator and container subsume the notion of *state* which can change when containers are searched or updated. The modeling of the state by classical algebraic specifications involves extra data structures representing the state which must be explicitly passed to a function as argument and yielded as result. Algebraic specifications are also not very convenient for describing a data structure with an arbitrary order of component insertion and deletion. This leads to very complex specifications with problems of describing the differences between different types of containers.

At the same time it seems very natural to consider containers and iterators as objects possessing state and define container classes and iterator classes as generic object types parameterized by the component type. To define formally iterator categories and thus represent the hierarchy of iterator classes, we need a notion of *class category* similar to that of sort classes of Spectrum. The elaboration of such a notion is one of the tasks of this paper.

The paper is organized in the following way.

Concrete object types defining sets of states of potentially mutable objects and object-oriented dynamic systems generalizing the communities of object states and their transitions are introduced in Section 2. Unconstrained generic object types, functions and procedures are defined in Section 3. Generic vector types and list types as typical representatives of Standard Template Library are specified in Section 4 and Section 5, respectively. Object type categories permitting the classification of object types on the base of their operations are introduced in Section 6. Generic object types, functions and procedures constrained by type classes are defined in Section 7. Some related work is discussed in Section 8, and some conclusions are given in Section 9.

It is assumed that the reader is familiar with the basic ASMs notions which can be found in [1, 2]. The familiarity with the formal aspects of object-oriented ASMs [3] is desirable, but not obligatory.

2. CONCRETE OBJECT TYPES

2.1. Instance algebras

We distinguish between *data* and *objects* and, respectively, between *data types* and *object types*. A data identifies itself and is immutable. An object possesses a unique identifier and state which can be mutable. A data type defines a set of immutable values and operations over them. An object type defines a set of states of a potentially mutable object and a number of methods capable to observe and update the object's state.

We consider that data types are specified by means of one of the algebraic specification languages like Spectrum [5], Ruslan [6] or CASL [7]. Let Σ' be the signature of a number of data type specifications, we call it *data signature* in the sequel. An *object- structured signature* Σ extends Σ' with a number of *object type signatures* [3]. An *object type specification* over Σ' is a pair <*object type signature, axioms*>. Let $OTYPE$ be a set of (object type) names and $O\Phi$ a set of object type specifications. An *object-structured specification* is then a triple $< OTYPE, O\Phi, int^o >$, where int^o is a function mapping $OTYPE$ into $O\Phi$. If $T \in OTYPE$, $ots \in O\Phi$, and $int^o(T) = ots$, then the maplit $< T \mapsto ots >$ is the specification of the object type T (in this case we also write sometimes that ots is marked with T).

An object type specification consists of an *object type signature* and *axioms* in the following form:

class Object-type-name = **spec**
[object-type-signature]
{axioms},

where *object type signature* is a triple *set-of-attribute-signatures; set-of-mutator-signatures; set-of-observer-signatures.*

An attribute or observer signature has the following form: *operation-name: operation-profile*; where *operation-profile* is either T or $T_1, \ldots, T_n \longrightarrow T$, where T, T_i are data/object type names indicating the types of attribute/observer parameters (if any) and result. A mutator signature is either just a mutator name or *mutator-name: mutator-profile*, where *mutator-profile* is a sequence of data/object type names indicating the types of mutator parameters.

Intuitively, a tuple of attributes defines an object's state, an observer is a function computing something at a given object's state, and a mutator is a procedure changing an object's state. Attributes are often called *instance variables*, and observers and mutators are often called *methods* in object-oriented programming languages.

Notation: in the following examples sets of attribute, observer, and mutator signatures in the object type signature are preceded by keywords **attribute**, **observer**, and **mutator**, respectively.

As an example, let us consider a simple model of memory consisting of locations storing integer numbers. A consecutive number of locations form a vector which is a structure allowing both sequential and random access to its elements. A location representing a vector element is called a *vector iterator* in the sequel.

Such an iterator can be understood as an object possessing the attribute *value_stored* updated by the mutator *put_value* and observed by the observer *get_value*. Since we wish to provide both sequential and random access to the vector elements, we define several extra observers as well. As a result, we create the following object type signature:

class VecIt = **spec**
[**attribute** value_stored: Integer;
mutator put_value: Integer;
observer get_value: Integer;
advance, retreat: VecIt;
plus, minus: Nat $\longrightarrow$ VecIt;
difference: VecIt $\longrightarrow$ Nat;
eq, neq, less, greater, leq, geq: VecIt $\longrightarrow$ Boolean]

In the above example, the community of vector elements is represented by the class VecIt (vector iterator) with several methods permitting to move either to the next (*advance*) or previous (*retreat*) element, to jump several elements forward (*plus*) or backward (*minus*), to calculate the distance between two elements (*difference*) and to compare the element (location) identifiers (*eq, neq, less, greater, leq, geq*).

In a particular algebra A, a set of elements called *object identifiers* is associated with an object type name T, and a (partial) function $at_T^A : A_T \longrightarrow (A_{T'_1}, \ldots, A_{T'_n} \longrightarrow A_{T'})$[1] is associated with the attribute name $at : T'_1, \ldots, T'_n \longrightarrow T'$ (a function $at_T^A : A_T \longrightarrow A_{T'}$ is associated with the attribute name $at : T'$) in an object type signature marked with T; such a function is called an *attribute function.* If $id \in A_T$, then $at_T^A(id)$ is an *attribute* of id.

Thus, in any algebra A, each object of the type VecIt is supplied with a unique identifier, and a function $value\_stored^A$ mapping object identifiers to integer numbers is created.

[1]For a data/object type T, A_T denotes the set associated with T in the algebra A.

Such an algebra extending the algebra of data types is called an *instance algebra.* It represents some state of a number of objects. 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.* An update of an object's state as well as creation or deletion of an object leads to the transformation of one instance algebra into another.

To define the interpretation of observer and mutator names, a notion of *dynamic system* is introduced.

2.2. Dynamic system

We discussed above only functions defined inside the frames of object types. In a more general case, an instance algebra can possess a number of "independent" functions and constants (i.e., functions without arguments) defined outside of an 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* $D\Sigma$ as an extension of an object-structured signature as follows: $D\Sigma = < \Sigma, DF >$, where Σ is an object-structured signature and DF is set of function and procedure signatures defined in the same way as attribute and mutator signatures are, respectively, defined above.

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

Examples.

dynamic const an_iterator: VecIt;
dynamic function matrix: Nat, Nat $\longrightarrow$ VecIt;
proc allocate: Nat – *allocation of a number of iterators*;

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 name are interpreted by invocations of the corresponding constants or functions. In the sequel, mentioning an instance algebra A, we mean a $D\Sigma$-algebra.

Let now OID be a set of object identifiers and $|D(A')|$ a set of instance algebras satisfying the following conditions:

- all algebras in $|D(A')|$ have the same data algebra A', and

- if T is an object type name 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).

Given a dynamic system signature $D\Sigma = < \Sigma, DF >$, a set of object identifiers OID, and a set of instance algebras $|D(A')|$, we associate:

- with each observer signature $b : T_1, \dots, T_n \longrightarrow T'$ 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 $< A, < id, a_1, \dots, a_n >>$, where $A \in |D(A')|$, $id \in A_T$, and $a_i \in A_{T_i}, i = 1, \dots, n$;
- with each mutator signature $m : T_1, \dots, T_n$ in an object type signature marked with T, a partial map (called a *mutator*), $m_T^{D(A')}$, associating an algebra $B \in |D(A')|$ with each pair $< A, < id, a_1, \dots, a_n >>$, where $A \in |D(A')|$, $id \in A_T$, and $a_i \in A_{T_i}, i = 1, \dots, n$.
- with each procedure signature $p : T_1, \dots, T_n$ from DF, a partial map (called a *procedure*), $p_T^{D(A')}$, associating an algebra $B \in |D(A')|$ with each pair $< A, < a_1, \dots, a_n >>$, where $A \in |D(A')|$, and $a_i \in A_{T_i}, i = 1, \dots, n$.

Thus, for the mutator *put_ value* defined in the class $VecIt$, a function mapping states to states is created, and for each observer defined in the class, a function mapping object identifiers and states to the values of the types indicated is created. For example, the function *advance* may produce the identifier considered to be next to a given identifier in a given state. The procedure associated with *allocate* will produce a new state by attaching a number of locations to the previous state.

An object-oriented dynamic system $DS(A')$ of signature $D\Sigma$ consists of a set of object identifiers OID, a set of instance algebras $|D(A')|$, and a set of observers, mutators, and procedures defined above.

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 $D(A')$.

2.3. 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 (transition rules are special cases of transition terms). The main rules for creating the terms are following (we omit interpretation where it is self-evident).

1. If $at : T_1, \ldots, T_n \longrightarrow T'$ is an attribute/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 $t.at(t_1, \ldots, t_n)$ is a term of type T'. Examples: $id.value\_stored$, $id.advance$, $id.plus(2)$, $id.less(id1)$. The term is interpreted by invoking the corresponding attribute/observer function in a given state A.
2. 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. The interpretation of this term allows us to check whether the argument term is defined in a given instance algebra. For example, the interpretation of $\mathbf{D}(id.advance)$ will let us know whether there is an identifier next to id in the current state.
3. If $m : T_1, ..., T_n$ is a mutator signature from the object type signature marked with T, $t_1, ..., t_n$ are terms of types $T_1, ..., T_n$, respectively, and t is a term of type T then $t.m(t_1, ..., t_n)$ is a transition term called a *mutator call.* Example: $id.put\_value(3)$. This kind of term serves for indicating an update of a mutable object. The term is interpreted by invoking the corresponding mutator function.
4. If $m : T_1, ..., T_n$ is a procedure signature, $t_1, ..., t_n$ are terms of types $T_1, ..., T_n$, respectively, then $t.m(t_1, ..., t_n)$ is a transition term called a *procedure call.* The term is interpreted by invoking the corresponding procedure with the given arguments.
5. If T is an object type name, f is the name of a function with the profile $T_1, \ldots, T_n \longrightarrow T$, $t_1, \ldots, t_n$ are terms of types $T_1, \ldots, T_n$, respectively, then $f(t_1, \ldots, t_n) := new(T)$ is a transition term. The interpretation of this transition term leads to the creation of a new object identifier with totally undefined attribute functions and forcing $f(t_1, \ldots, t_n)$ to produce this identifier.

2.4. Axioms

The specification of the method behavior is done by means of axioms.

An *axiom* is either a static axiom or a dynamic axiom. A *static axiom* is a pair $t_1 == t_2$, where t_1 and t_2 are two terms of the same type. A given algebra A satisfies a static axiom if the interpretation of both parts of the axiom in A produces the same result. A *dynamic axiom* has the same form, where t_1 and t_2 are transition terms which are generalizations of transition rules as described above. A given dynamic system $DS(A')$ satisfies a dynamic axiom if the interpretation of both parts of the axiom in any state $A \in |DS(A')|$ produces the same state B.

To specify the object type VecIt, we define 3 auxiliary dynamic functions:

vecit_number: Nat = 0;
natvec: Nat $\longrightarrow$ VecIt;
vecnat: VecIt $\longrightarrow$ Nat;

The first function indicates the number of vector iterators existing in the algebra, the second one maps natural numbers into vector iterators, and the third one maps vector iterators into natural numbers. Thus the functions imitate the relationship between memory addresses and natural numbers. Initially the first function is set to zero, the other two are undefined. Updates of the functions are produced by a special procedure *allocator* defined in Section 4. With the use of these functions, we can write the following axioms for the above VecIt signature (typical axioms for equality and non equality are omitted):

{**forall** i, i1: VecIt, n: Nat. – *declaration of universally quantified variables*
i.put_value(x) == i.value_stored := x; – *dynamic axiom*
i.get_value == i.value_stored; – *this one and all the other are static axioms*
i.advance == natvec(vecnat(i) + 1);
i.retreat == natvec(vecnat(i) – 1);
i.plus(n) == natvec(vecnat(i) + n);
i.minus(n) == natvec(vecnat(i) – n);
i.difference(i1) == vectnat(i) – vecnat(i1)
i.less(i1) == vectnat(i) < vecnat(i1);
i.greater(i1) == vectnat(i) > vecnat(i1);
i.leq(i1) == vectnat(i) ≤ vecnat(i1);
i.geq(i1) == vectnat(i) ≥ vecnat(i1)}

3. GENERIC OBJECT TYPES

We cannot be satisfied with only concrete object types because many object types can have a similar structure and it would be tiresome to specify them again and again. Therefore a notion of generic object type is introduced. We start with the simplest case, *unconstrained generic object types.*

An *object-structured specification* Σ is defined above as a triple $< OTYPE, O\Phi, int^o >$, where int^o is a function mapping $OTYPE$ into $O\Phi$. We propose the following way of constructing the names in $OTYPE$ (which will be called *type terms* from now on), using two non intersecting sets of names, $\underline{S}$ and $\underline{R}$: if $S \in \underline{S}$, then S is a type term; if $T_1, ..., T_n$ are type terms and $R \in \underline{R}$, then $R(T_1, ..., T_n)$ is a type term.

Let now $int^o(R(T_{11}, ..., T_{1n})) = Spec1$ and $int^o(R(T_{21}, ..., T_{2n})) = Spec2$, where $R(T_{11}, ..., T_{1n})$ and $R(T_{21}, ..., t_{2n})$ are type terms and $Spec1$ and $Spec2$ are object type specifications. We say that the object type $R(T_{11}, ..., T_{1n})$ is a *sibling* of the object type $R(T_{21}, ..., T_{2n})$ if the replacement of each T_{1i} with T_{2i}, $i = 1, ..., n$, in $Spec1$ converts it into $Spec2$. We can propose a special way of constructing a part of the function int^o for a family of object type siblings.

Let $q_1, ..., q_k$ be names (of type parameters). A pair $< R(q_1, ..., q_k), Spec >$ (where $R \in \underline{R}$ and $Spec$ is an object type specification additionally using $q_1, ..., q_k$ as type terms in method signatures) is part of the function int^o, such that for any type term T_i, $i = 1, ..., k$, $int^o(R(T_1, ..., T_k)) = Spec[q_1/T_1, ..., q_k/T_k]$, where $Spec[q_1/T_1, ..., q_k/T_k]$ is an object type specification produced by replacing each q_i in $Spec$ with T_i.

A pair $< R(q_1, ..., q_k), Spec >$ is called a *generic type specification*, and R is called a *generic object type.* The replacement of type parameters with type terms in both parts of a generic type specification is called a *generic type instantiation.* Note that due to the use of the function int^o, we do not need to introduce a special semantics for generic object types. A generic type specification in this approach is just a way of defining a part of this function. This corresponds one to one to the practice of modern programming languages (like C++) regarding generic object types as templates.

Two type terms $R(T_{11}, ..., T_{1n})$ and $R(T_{21}, ..., T_{2n})$ are equivalent if T_{1i} and T_{2i}, $i = 1, ..., n$, are the same type name or if they are equivalent.

Example. Taking into account that in a real memory we would like to have locations storing values of different types, we could define the following signature of a generic vector iterator type:

class VecIt(T) = **spec**

[**attribute** value_stored: T;
mutator put_value: T;
observer get_value: T;
advance, retreat: VecIt(T);
plus, minus: Nat $\longrightarrow$ VecIt(T);
difference: VecIt(T) $\longrightarrow$ Nat;
eq, neq, less, greater, leq, geq: VecIt(T) $\longrightarrow$ Boolean]

To write axioms for this generic type, we apparently must have generic versions of the above functions *vecit_number*, *natvec*, and *vecnat*. Thus, we introduce some notions permitting us to define generic functions:

- a *generic function profile* is a pair $< (q1, ..., qk), FP^q >$, where $q1, ..., qk$ are names (of type parameters) and FP^q is a function profile constructed by extending the set of type terms with $q1, ..., qk$;
- a *generic function signature* is a pair $op : FP^q$, where *op* is an operator and FP^q is a generic function profile.

Notation: we use the brackets **gen** ... **profile** to embrace type parameters.

Examples:

vecit_number: **gen** T **profile** Nat = 0;
natvec: **gen** T **profile** Nat $\longrightarrow$ VecIt(T);
vecnat: **gen** T **profile** VecIt(T) $\longrightarrow$ Nat;

If $op :< (q1, ..., qk), FP^q >$ is a generic function signature and $T1, ...Tk$ are type terms , then $op(T1, ..., Tk) : FP$ is an instantiated function signature, where FP is a function profile obtained from FP^q by replacing each qi with Ti; $op(T1, ..., Tk)$ is called an *instantiated operator*. For example, *natvec(Integer)* is an instantiated operator with the profile $Nat \longrightarrow VecIt(Integer)$. Instantiated operators are used for producing data terms in the same way as ordinary operators do, for example, *natvec(Integer)(2)*.

According to the extension of the object-structured signature with generic function signatures, an algebra A of a given signature is extended with a set of functions map^{Aq}, one for each generic function signature $op :< (q1, ..., qk), FP^q >$; such a function binds an instantiated operator $op(T1, ..., Tk)$ to a function in A.

Notation: instead of $op(T11, ..., T1k)(t1, ..., tn)$, we write $op(t1, ..., tn)$ where it seems appropriate.

A generic function specification consists of a generic function signature and a set of axioms.

Example. The axioms of the generic object type VecIt(T) would be now the following:

{**forall** T: TYPE, i, i1: VecIt(T), n: Nat.
i.put_value(x) == i.value_stored := x; – *dynamic axiom*
i.get_value == i.value_stored; – *this one and all the other are object axioms*
i.advance == natvec(T)(vecnat(T)(i) + 1);
i.retreat == natvec(T)(vecnat(T)(i) – 1);
i.plus(n) == natvec(T)(vecnat(T)(i) + n);
i.minus(n) == natvec(T)(vecnat(T)(i) – n);
i.difference(i1) == vectnat(T)(i) – vecnat(T)(i1)
i.less(i1) == vectnat(T)(i) < vecnat(T)(i1);
i.greater(i1) == vectnat(T)(i) > vecnat(T)(i1);
i.leq(i1) == vectnat(T)(i) ≤ vecnat(T)(i1);
i.geq(i1) == vectnat(T)(i) ≥ vecnat(T)(i1)}

Similar to generic functions, generic procedures can be defined:

- a *generic procedure profile* is a pair $< (q1, ..., qk), PP^q >$, where $q1, ...,$ qk are names (of type parameters) and PP^q is a procedure profile constructed by extending the set of type terms with $q1, ..., qk$;
- a *generic procedure signature* is a pair $p : PP^q$, where p is a procedure name and PP^q is a generic procedure profile.

If $p :< (q1, ..., qk), PP^q >$ is a generic procedure signature and $T1, ...Tk$ are type terms , then $p(T1, ..., Tk) : PP$ is an instantiated procedure signature, where PP is a procedure profile obtained from PP^q by replacing each qi with Ti; $p(T1, ..., Tk)$ is called an *instantiated procedure name.* Instantiated procedure names are used for producing transition terms in the same way as ordinary procedure names do.

According to the extension of the dynamic system signature with generic procedure signatures, a dynamic system $DS(A')$ of a given signature is extended with a set of functions map^{DSq}, one for each generic procedure signature $p :< (q1, ..., qk), PP^q >$; such a function binds an instantiated procedure name $p(T1, ..., Tk)$ to a procedure in $DS(A')$.

A generic procedure specification consists of a generic procedure signature and a set of dynamic axioms. An example of a procedure specification can be found at the end of the next section.

4. GENERIC VECTOR TYPES

We can now give a formal specification of the generic vector type informally defined in [4]. It is assumed that vector elements (which are vector iterators) are numbered starting from zero. The clause **dom** in the specification indicates the domain of a partial function: in a domain specification **dom** $t : b$, t is defined if and only if b evaluates to *true*. A transition rule of the form **set** $R_1, ..., R_n$ **end** indicates parallel execution of the transition terms $R_1, ..., R_n$.

```
class Vector(T) = spec
 [attribute comp: Nat ⟶ VecIt(T);
            – vector's components, initialized by an allocator
     size: Nat; – vector's current size, initialized by an allocator
     max_size: Nat; – vector's maximum size, initialized by an allocator
     begin: VecIt(T); – the first element of a vector, initialized by an allocator
  mutator empty_vec; – default constructor
     intialized_vec: Nat, T; – initialization of the first n components
     copy: Vector(T); – copy constructor
     push_back: T; – append an element at the end of a vector
     pop_back; – delete the last vector's element
     insert: VecIt(T), T; – insert an element at the position indicated
     erase: VecIt(T); – remove the element indicated
     swap: Vector(T); - swap the contents of two vectors
  observer empty: Boolean; – is a vector empty?
     []: Nat ⟶ T; – fetch a vector's element
     front, back: T; – first and last vector's elements
     end: VecIt(T)]
            – the identifier of the element following the last vector's element
 {forall x, x1: T, iv, iv1: VecIt(T), n, n1: Nat, v, v1: Vector(T).
  dom v[n]: n ≥ 0 & n ≤ v.size;
  dom v.intialized_vec (n, x): n ≤ v.max_size;
  dom v.push_back(x): v.size < v.max_size;
  dom v.insert(iv, x): v.size < v.max_size;
  dom v.erase(iv): iv.geq(v.begin) & iv.less(v.end) & v.size > 0;
  v.empty_vec == v.size := 0;
  v.intialized_vec (n, x) == set forall i: Nat.
            if i < n then v.comp(i).value_stored := x, v.size := n end;
  v.copy(v1) == v := v1;
  v.push_back(x) ==
            set v.comp(v.size).value_stored := x; v.size := v.size + 1 end;
  v.pop_back == v.size := v.size - 1;
  v.insert(iv, x) == forall i: Nat. set v.size := v.size + 1,
        if v.comp(i).geq(iv) & i < v.size
        then v.comp(i+1).value_stored := v.comp(i).value_stored,
        if v.comp(i).eq(iv) then v.comp(i).value_stored:= x end;
  v.erase(iv) == set forall i: Nat. if v.comp(i).geq(iv) & i <= v.size
        then v.comp(i).value_stored := v.comp(i+1).value_stored,
        v.size := v.size - 1 end;
  v.empty == v.size =0;
  v[n] == v.comp(n).value_stored;
  v.end == v.begin.plus(v.size+1);
  v.front == v.comp(0).value_stored;
  v.back == v.comp(v.size).value_stored }.
```

A very interesting task is the specification of a vector allocator, a special procedure which allocate memory for a particular vector (according to STL, such a procedure is associated with each container class, we define it here for vectors). A new kind of transition rules, *For-loop*, based on the sequential transition rule defined in [3] is used in the specification. This rule is defined as follows:

Let i be a variable of type Nat, t_1 and t_2 be terms of type Nat not containing i, and R be a rule, then

for $i = t_1$ **to** t_2 **do** R

is a transition rule called *For-loop*. Interpretation in an algebra A:

(**for** $i = t_1$ **to** t_2 **do** $R)^A$ =

(**seq** $R[t_1/i]$, **if** $t_1 < t_2$ **then for** $i = t_1 + 1$ **to** t_2 **do** $R)^A$.

The allocator can now be defined in the following way:

```
proc vec_allocator: gen T profile T, Nat
{forall v: Vector(T), n: Nat.
 vec_allocator(T)(n) == seq
  set v.max_size := n, v.size := 0,
        for i = 0 to n do v.comp(i) := new_var(Vector(T)
        v.begin := v.comp(0)
  end,
  forall i: Nat. if i ≤ n then
  set vecit_number(T) := vecit_number(T) + n,
        vecnat(T)(v.comp(i)) := vecit_number(T) + i,
        natvec(T)(vecit_number(T) + i) := v.comp(i)
  end end};
```

5. LIST TYPES AND ITERATORS

A list is a sequence of elements ordered according to the use of constructor operations "push_back" (appends an element to the end of a list), "push_front" (appends an element to the beginning of a list), and "insert" (insert an element in the middle of a list). All list elements are numbered starting with one for the first element. The number of the last element is equal to the number of list's elements. A list element is an object of the corresponding iterator type ListIt. We define it first.

```
class ListIt(T) = spec
 [attribute value_stored: T;
        pred, next: ListIt(T);
 mutator put_value: T;
 observer get_value: T;
        advance, retreat: VecIt(T);
        eq, neq: VecIt(T) ⟶ Boolean]
 {forall i: ListIt(T), x: T.
  i.put_value(x) == i.value_stored := x;
  i.get_value == i.value_stored;
  i.advance == i.next;
  i.retreat == i.pred}.
```

Now we can define the generic list type. Due to space limitations, the specification of some mutators is left to the reader. Note that a transition rule of the form **seq** $R_1, ..., R_n$ **end** indicates sequential execution of the transition terms $R_1, ..., R_n$, and a transition rule of the form **while** b **do** R **end** indicates the repetition of the execution of the transition term R. Formal semantics of both rules can be found in [3].

```
class List(T) == spec
 [attribute begin, end: ListIt(T);
        size: Nat; – list current size
  mutator empty_list; – empty list constructor
        intialized_list: Nat, T; – initialized list constructor
        copy: List(T); – copy constructor
        push_front: T; – append an element at the beginning of a list
        push_back: T; – append an element at the end of a list
        pop_front; – delete the first list's element
        pop_back; – delete the last list's element
        insert1: ListIt(T), T; – insert one element at the position indicated
```

```
          insertN: ListIt(T), Nat, T; – insert N elements at the position indicated
          erase1: ListIt(T); – remove the element indicated
          eraseN: ListIt(T), ListIt(T); – remove the elements between the iterators
  observ empty: Boolean; – is the list empty?
          front, back: T] – first and last list's elements
 {forall x, x1: T, n, n1: Nat, l, l1: List(T), i, i1: ListIt(T).
  dom l.pop_front: l.size > 0;
  dom l.pop_back: l.size > 0;
  dom l.erase1(i): l.size > 0 & i.geq(l.begin) & i.less(l.end);
  dom l.eraseN(i, i1): l.size > 0 & i.geq(l.begin) & i1.leq(l.end) & i.less(i1);
  dom l.front: l.size > 0;
  dom l.back: l.size > 0;
  l.empty_list == set l.begin := undef, l.end := undef, l.size := 0 end ;
  l.push_back(x) == import temp: ListIt(T) in
          set temp.pred := l.end, temp.next := undef, temp.value_stored := x,
                  l.end := temp, l.size := l.size +1,
                  if l.size = 0 then l.begin := temp else l.end.next := temp end ;
  l.intialized_list(n, x) ==
                  seq l.empty_list, while l.size < n do l.push_back(x) end ;
  l.copy(l1) == seq sequence of transition rules end;
  l.push_front(x) == import temp: ListIt(T) in set of transition rules;
  l.pop_front == if l.size > 1 then
          set l.begin := l.begin.next, l.begin.next.pred := undef, size := size -1 end
          else set l.begin := undef; l.end := undef; l.size := 0 end ;
  l.pop_back == if l.size > 1 then set of rules else set of rules;
  l.insert1(i, x) == if ¬D(i.next) then set of rules else set of rules;
  l.insertN(i, n, x) ==
                  seq l.insert1(i, x), if n > 1 then l.insertN(i.next, n-1, x) end ;
  l.erase1(i) == if ¬D(i.next) then l.pop_back – last element is deleted
          elseif ¬D(i.pred) then l.pop_front – first element is deleted
          else set i.pred.next := i.next, i.next.pred:= i.pred, size := size - 1 end ;
  l.erase(i, i1) == seq l.erase(i), if i.next.neq(i1) then l.erase(i.next, i1) end ;
  l.front == l.begin.value_stored; l.back == l.end.value_stored}.
```

A list allocator just initializes the attributes of a list, its specification is trivial.

6. OBJECT TYPE CATEGORIES

The generic constructs defined above permit us to specify algorithms over data structures, such as vectors or lists, abstracted from the type of the component of the structure. We would like also to specify algorithms abstracted from the data structure itself, i.e. to specify an algorithm capable, for example, to manipulate both vectors and lists. *Object type categories* serve this purpose.

An object type category resembles a *sort class* of Spectrum [5] or *type class* of Ruslan [6] both based on type classes introduced in [8] and sypes introduced in [9]. It defines a set of object type specifications with some common properties.

Let CAT be a set of names and CS be a set of specifications constructed like object type specifications with the use of an extra object type name "@", and $int^c : CAT \longrightarrow CS$ be a function mapping names in CAT to specifications in CS. If $C \in CAT$, $cs \in CS$, and $cs = int^c(C)$, then the maplet $< C \mapsto cs >$ is the specification of the object type category C.

Let $< C \mapsto cs >$ be the specification of an object type category and $< T \mapsto ots >$ be the specification of an object type. It is said that the object type T is a type of the category C (or T belongs to C) if $cs[T/@] \in ots]$, where $cs[T/@]$ is the specification cs with the symbol "@" replaced with T.
Example:

```
classcat Equal = spec — category of object types with equality operation
 [observer eq, neq: @ ⟶ Boolean]
 {forall x, y: @, exist z: @.
  x.eq(x) == true; x.eq(y) == y.eq(x);
  x.eq(z) & z.eq(y) == x.eq(y);
  x.neq(y) == ¬x.eq(y)}.
```

Any object type possessing the operations *eq* and *neq* specified as above is the type of the category Equal.

Constructing a type category, we can inherit the specification of an existing type category producing the union of the specifications as result.

Like an object type, an object type category can be generic, i.e., it can use type parameters in its specification. The definition of a generic object type category is the same as the definition of a generic object type. Iterator categories serve as examples.

1. Each iterator type of the following category has operations *advance*

and *get_ value* in addition to the operations of the category "Equal".[2]

classcat InputIterator(T) = **spec** Equal
 [**observer** advance: @; – *advances an iterator one position forward*
 get_value: T] – *dereferencing operation for reading*

2. Each iterator type of the following category has operations *advance* and *put_ value* in addition to the operations of the category "Equal".[3]

classcat OutputIterator(T) **spec** Equal
 [**observer** advance: @; – *advances an iterator one position forward*
 mutator put_value: T] – *stores a new value in an iterator*

3. Each iterator type of the following category has a mutator *put_ value* in addition to the operations of the category "InputIterator".

classcat ForwardIterator(T) = **spec** InputIterator(T)
 [**mutator** put_value: T]

4. Each iterator type of the following category has the operation *retreat* in addition to the operations of the category "ForwardIterator".

classcat BidirectionalIterator(T: TYPE) = **spec** ForwardIterator(T)
 [**observer** retreat: @] - *advances an iterator one position backward*

5. Each iterator type of the following category has several operations in addition to the operations of the category "BidirectionalIterator".

classcat RandomAccessIterator(T) = **spec** BidirectionalIterator(T)
 [**observer** plus, minus: Nat $\longrightarrow$ @;
 difference: @ $\longrightarrow$ Nat;
 less, greater, leq, geq: @ $\longrightarrow$ Boolean]

According to the definitions, an object type VecIt(T) belongs to the type categories RandomAccessIterator(T), BidirectionalIterator(T), ForwardIterator(T), OutputIterator(T), InputIterator(T), and Equal. An object type ListIt(T) belongs to the type categories BidirectionalIterator(T), ForwardIterator(T), OutputIterator(T), InputIterator(T), and Equal (it does not belong to the type category RandomAccessIterator(T), however). Thus, a vector iterator can be used in any algorithm requiring either a random access iterator or bidirectional iterator or forward iterator or input iterator or output iterator. In the same way a list iterator can be used in any algorithm except that one which requires a random access iterator.

[2]the category of input iterators is introduced in STL to allow iterating over input streams in the same way as, say, over vectors.

[3]the category of output iterators is introduced in STL to allow iterating over output streams in the same way as, say, over vectors.

7. CONSTRAINED GENERICITY

According to the definitions of generic components in Section 3, any type can be used as instantiation argument. At the same time, it is needed very often that only a type belonging to a certain type category could be substituted. Therefore, the definitions of generic components should be changed with taking into account this constraint. The definition of a constrained generic object type is now the following.

Let $q_1 : C_1, ..., q_k : C_k$ be names (of type parameters) indexed with type category names. A pair $< R(q_1 : C_1, ..., q_k : C_k), Spec >$ (where $R \in \underline{R}$ and *Spec* is an object type specification additionally using $q_1, ..., q_k$ as type terms in operation signatures and operators from $C_1, ..., C_k$ in axioms) is part of the function int^o, such that for any type term T_i of type category C_i, $i = 1, ..., k$, $int^o(R(T_1, ..., T_k)) = Spec[q_1/T_1, ..., q_k/T_k]$, where $Spec[q_1/T_1, ..., q_k/T_k]$ is an object type specification produced by replacing each q_i in *Spec* with T_i.

The definition of the constrained generic type category is done in a similar way. Constrained generic functions are defined as follows.

1. A *generic function profile* is a pair $< (q1 : C1, ..., qk : Ck), FP^q >$, where $q1 : C1, ..., qk : Ck$ are names (of type parameters) indexed with type category names and FP^q is a function profile constructed by extending the set of type terms of type categories $C1, ..., Ck$ with $q1, ..., qk$, respectively.
2. A *generic function signature* is a pair $op : GFP$, where *op* is an operator and GFP is a generic function profile.

If $op^q :< (q1 : C1, ..., qk : Ck), FP^q >$ is a generic function signature and $T1, ...Tk$ are type terms such that each $Ti, i = 1, ..., k$ belongs to the type category Ci, then $op^q(T1, ..., Tk) : FP$ is an instantiated function signature, where FP is a function profile obtained from FP^q by replacing each qi with Ti.

Examples:

1. Specify a function which looks for an element in a data structure between $first$ and $last$ iterators and returns the iterator storing the value if it is found and the last iterator if the value is not found.

```
func find: gen I: InputIterator, T: TYPE profile: I, I, T ⟶ I;
{forall first, last: I, value: T.
 find(first, last, value) ==
  if first.get_value = value ∨ first = last then first
 else find(first.advance, last, value)}.
```

Now, if we have the following declarations:

```
    vec: Vector(Integer);
    list: List(Char);
```

we can invoke the function in the folowing ways:

```
    find(vec.begin, vec.end, 7);
    find(list.begin, list.end, 'a');
```

In this case both vector iterators and list iterators can be used because both belong to the category of input iterators required in the function specification. In the next example list iterators cannot be used.

2. Specify a function which performs binary search in a structure containing ordered components and returns the identifier of the iterator containing the element to be found (for simplicity we assume that the structure really contains the element).

```
func binary_find: gen I: RandomAccessIterator, T: Ordered profile I, I, T ⟶ I;
{forall first, last: I, value: T.
 binary_find(first, last, value) == let d = last.difference(first), h = d/2,
                                        current = first.plus(h), cv = current.get_value in
  if cv = value then current
  elseif value < cv then binari_find(first, current, value)
  else binary_find(current, last, value)}
```

Now, we can call the function with vector iterators like the following:

```
    binary_find(vec.begin, vec.end, 7)
```

because these iterators belong to the class RandomAccessIterator, and cannot call it with list iterators. Note the use of the type class Ordered (not defined here) needed to make sure that the type of the components contains the operation "<".

8. RELATED WORK

We are not going to discuss here the approaches representing object states as elements of the same algebra. The works along this approach are heavily based on traditional algebraic specification methods. We can only repeat after F. Parisi-Presicce and A. Pierantonio that "the algebraic framework so far has been inadequate in describing the dynamic properties of objects and their state transformation as well as more complex notions typical of the object oriented paradigm such as object identity and persistency of objects" [10]. The interested reader can refer to [11, 12, 13, 14].

We review here several works considering object states as algebras. The first of them are object-oriented extensions of the prominent specification methods VDM [15] and Z [16]. These are VDM++, Object-Z, and Z++.

VDM++ [17] introduces the notion of a class definition as a template for a collection of objects possessing a number of instance variables (internal state) and methods (external protocol). The definitions of the methods of existing classes can be inherited when a new class is defined (multiple inheritance). Object's initial state and invariant can be specified. A set of statements typical of the imperative programming language is provided. Unfortunately, the description of the semantics of the language is done rather informally, in the way the semantics of programming languages is usually done. As a result, the user gets the impression that VDM++ is a programming language provided with some specification facilities (pre- and post-conditions) rather than a specification language. No genericity is provided in the language.

Object-Z [18] practically has the same features as VDM++ with the difference that it is based on Z. A class is here a set of attributes and a set of operations acting upon these attributes. In contrast to VDM++, the semantics of Object-Z is formally given. The state is considered here as a function from a set of identifiers (attributes) to the set of all possible values. One can say that it corresponds to a homogeneous algebra whose signature contains only constant symbols (compare it with a more general case of ASM where functions as algebra components play significant role). An association of an operation identifier with a finite partial function defining the values of the operation arguments is called an event. A state transition is defined as a triple consisting of an event and two states (source and target). A class definition can be supplied with a number of type parameters, a kind of unconstrained genericity is provided in this way. No notion of class category and, respectively, constrained genericity exists in the language. Object creation is also not provided by the language. Thus, a specification like that one of lists given above is not possible.

Z++ [19, 20] is another development based on Z providing facilities comparable to those of Object-Z. The main difference is that its syntax is quite different from that of Object-Z (which actually follows the syntax of Z) and is chosen to stress the commonalties with of the concepts of the language with those of object-oriented programming languages like Effel. Formal semantics of the language is defined by means of category theory. However, the main attention is paid to the formal definition of refinements between classes while such important notions as state, class, object, etc. are considered well-known and not defined in the semantics. Like in Object-Z, a class definition in Z++ can be supplied with a number of type parameters providing unconstrained genericity. Again, no notion of class category (in our sense) and, respectively, constrained genericity exists in the language. The semantics of a generic class specification is also not reported.

In [21] objects are elements of instance structures which are quadruples of algebras of different signatures. Specifications of the algebras resemble traditional algebraic specifications. One of the algebras is extended with extra "state function symbols" mapping object identifiers to their values. Dynamic operations serving for object evolution are modeled by algebra morphisms. The author believes that the specification of these operation should have an imperative nature, but he does not suggest a method of specification. The approach is further formalized with a heavy use of category theory in [10]. In contrast to all of this, we represent the state by a single algebra, we believe that there are no necessity for state function symbols since user-defined observers perfectly serve for this purpose, and we suggest a concrete method of specification by means of transition rules.

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. This work combined with Meseguer’s rewriting logic [22] has served as basis of the dynamic aspects of the language CafeOBJ [23]. 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 [24].

The specification language Troll [25] should be mentioned as one of the main practical achievements in the field. 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 [26], 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. Although Troll possesses generic facilities, non of them is formalized in [26].

Finally, a fundamental work [27] formalizing bounded parametric polymorphism similar to our constrained genericity should be paid attention. Here genericity is constrained by allowing only those type arguments which are subtypes of the parameter type. It is evident that our approach is more general since not every type category can be defined as an object type. Another peculiarity of the work is that an object does not possess a unique identifier, it is just a tuple of methods, and object's updates are simulated by method overrides generally producing new objects.

9. CONCLUSION

The mechanisms for the specification of generic object types and type categories are introduced in the paper. With the use of these mechanisms, many generic algorithms abstracting from the type of the data structure being manipulated can be easily specified. The technique is applied to the specification of some components of the Standard Template Library for C++. The library thus specified can be easily adapted to another object-oriented language. The experience obtained in the process of the specification has proved the power of the technique. Its main features can be summarized as follows:

1. We represent immutable values by data types and specify them algebraically.
2. We represent mutable objects possessing states by object types and specify them by means of transition rules.
3. We define generic (data, object) types to abstract from the type of the component.
4. We define (data, object) type categories to abstract from the structure.
5. We define a generic algorithm by means of transition rules manipulating the objects thus specified.

Tools supporting this style of specification remain the subject of further work.

REFERENCES

1. Y. Gurevich. Evolving Algebras 1993: Lipary Guide. In: *Börger, E., ed., Specification and Validation Methods*, Oxford University Press, 1995, pp. 9-36.
2. Y. Gurevich. *May 1997 Draft of the ASM Guide*. University of Michigan (available electronically from http://www.eecs.umich.edu/gasm/).

3. A.V. Zamulin. *Object-Oriented Abstract State Machines.* Proc. ASM workshop, Magderburg, Germany, 21-22 September, 1998, pp. 1-21 (available electronically from http://www.eecs.umich.edu/gasm/).
4. D.R. Musser and A. Saini. STL Tutorial and Reference Guide. Addison-Wesley, 1996.
5. Broy, M., Facchi, C., Grosu, R., ea. *The Requirement and Design Specification Language Spectrum, An Informal Introduction, Version 1.0.* Technische Universitaet Muenchen, Institut fuer Informatik, April 1993.
6. 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.
7. P.D. Mosses. *CoFI: The Common framework Initiative for Algebraic Specification and Development.* In: M. Bidoit and M. Dauchet, eds., TAPSOFT'97: Theory and Practice of Software Development, LNCS, vol. 1214, pp. 115-137.
8. P. Wadler and S. Blott. *How to make ad-hoc polymorphism less ad-hoc.* Conf. Record of the 16th ACM Annual Symp. on Principles of Progr. Lang., Austin, Texas, January 1989.
9. Nakajima, R., Honda, M., and Nakahara, H. *Hierarchical Program Specification: a Many-sorted Logical Approach.* Acta Informatica 14, pp. 135-155 (1980).
10. F. Parisi-Presicce and A. Pierantonio. *Dynamic behavior of Object Systems.* In: Recent trends in Data Type Specification. LNCS, vol. 906, 1995, pp. 406-419.
11. J.A. Goguen and R. Diaconescu. *Towards an Algebraic Semantics for the Object Paradigm.* Recent Trends in Data Type Specification, LNCS, 1994, vol. 785, pp. 1-29.
12. H.-D. Ehrig and A. Sernadas. *Local Specification of Distributed Families of Sequential Objects.* In: Recent Trends in Data Type Specifications. LNCS, vol. 906, 1994, pp. 219-235.
13. F. Parisi-Presicce and A. Pierantonio. *An Algebraic Theory of Class Specification.* ACM Transactions on Software Engineering and Methodology, April 1994, vol. 3, No. 2, pp. 166-169.
14. C. Cristea. *Coalgebra Semantics for Hidden Algebra: Parameterised objects and Inheritance.* in: Recent Trends in Algebraic development Techniques. LNCS, vol. 1374, 1997, pp. 174-189.
15. C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall, 1990.
16. J. M. Spivey. *understanding Z. A specification language and its formal semantics.* Cambridge University Press, 1988.
17. E.H. D,rr and J. van Katwijk. *A Formal Specification Language for Object Oriented Designs.* In: Computer Systems and Engineering (Proceedings of CompEuro'92). IEEE Compute Society Press, 1992, pp. 214-219.
18. R. Duke, P. King, G.A. Rose, and G. Smith. *The Object-Z specification language.* In: T. Korson, V. Vaishnavi, and B. Meyer, eds., Technology of Object-Oriented Languages and Systems: TOOLS 5, Prentice hall, 1991, pp. 465-483.
19. K. Lano and H. Houghton, eds., *Object-Oriented Specification case Studies.* Prentice Hall (object-oriented series), 1994.
20. K. Lano and H. Houghton. *The Z++ manual.* October 1994. Available from ftp://theory.doc.ic.ac.uk/theory/papers/Lano/z++.ps.Z

21. A. Pierantonio. *Making Statics Dynamic.* In: G. Hommel, editor, Proc. International Workshop on Communication based Systems, Kluwer Academic Publishers, 1995, pp. 19-34.
22. J. Meseguer. *Conditional rewriting logic as a unified model of concurrency.* Theoretical Computer Science, vol. 96, No 1 (April 1992), pp. 73–155.
23. R. Diaconescu and K. Futatsugi. *CafeOBJ Report.*World Scientific Publishing Co. Pte. Ltd, AMAST series in Computing", vol. 6,1998.
24. J. Meseguer. *A logical theory of concurrent objects and its realization in the Mode language.* In: Research Directions in Concurrent Object- Oriented Programming. The MIT Press, Cambridge, Mass., 1993, pp. 314-390.
25. 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.
26. M. Gogolla and R. Herzig. *An Algebraic Semantics for the Object Specification Language TROLL-light.* In: Recent Trends in Data Type Specifications, LNCS, vol. 906, 1995, pp. 290– 306.
27. M. Abadi and L. Cardelli. *A Theory of Objects.* Springer- Verlag, 1996.

А. В. Замулин

РОДОВЫЕ СРЕДСТВА В ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ МАШИНАХ АБСТРАКТНЫХ СОСТОЯНИЙ

Препринт
60

Рукопись поступила в редакцию 22.05.1999
Рецензент И. Н. Скопин
Редактор Л. А. Карева

Подписано в печать 26.05.1999
Формат бумаги 60×84 1/16 Объем 1,6 уч.-изд.л., 1,7 п.л.
Тираж 50 экз.

Отпечатано на ризографе "AL Group" 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6