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

Kirill O. Senoshenko

OBJECT-ORIENTED SPECIFICATIONS: SET-THEORY BASED AND ALGEBRAIC APPROACHES. A REVIEW

Preprint
91

Novosibirsk 2002

In this review, the main trends in object-oriented dynamic system specification are discussed. A classification for the specification approaches is given basing on the underlying model of system state; several representatives from different categories are presented. For each approach, a special attention is given to the completeness of support of various OO concepts, ability to define not only static aspects of the system but also its behavior and the specification transparency. Finally, a comparative review of the selected languages and formalisms is provided and directions for further work are outlined.

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

Сеношенко К. О.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ СПЕЦИФИКАЦИИ: ТЕОРЕТИКО-МНОЖЕСТВЕННЫЙ И АЛГЕБРАИЧЕСКИЙ ПОДХОДЫ. ОБЗОР

Препринт
91

Новосибирск 2002

Рассматриваются основные методы построения объектно-ориентированных спецификаций для динамических систем. Предлагается классификация этих методов, базирующаяся на внутренней модели состояния системы; рассматривается несколько языков спецификаций разных классов. При рассмотрении каждого языка внимание уделяется полноте поддержки объектно-ориентированного подхода, возможности описания не только статических свойств, но и динамического поведения системы и прозрачности спецификации. В завершении приводится сравнительный анализ выбранных языков и методов и намечаются направления дальнейшей работы.

1 INTRODUCTION

The aim of this paper is to give a comparative review of various approaches to object-oriented formal specifications.[*)] Object orientation is well-adopted for specifying both static and dynamic properties of complex scalable systems. Benefits of this methodology are widely recognized since it proposes techniques to enhance quality factors of software, such as *reusability*, *extensibility* and *compatibility*. Therefore, there are numerous languages and tools supporting it through the entire development cycle of a system, including program specification and validation stage.

Some of object-based specification languages extend existing formalisms with *data encapsulation*, *inheritance*, *polymorphism*, *subtyping* and other well-known key features of the object-oriented paradigm; others are more or less independent. Conventionally, these formalisms can be divided into two classes: formalisms in the first class do not have an underlying notion of a state (the so-called *pure algebraic approach* is most well-known among them) while those in the second class do support it. There are several models of a state; to mention only a few, these are:

- *first-order logic and set theory* (a state is a family of sets and bags),
- the *algebraic approach* (a state is an algebra),
- Petri nets / algebraic nets (a state is a net).

There are representatives for each of these groups [1,2] that additionally involve the temporal logic in describing dynamic properties of a system though it is not always the case.

Because of the limited size of this review, only the languages from the first two groups, which are most widely used nowadays, are discussed below. Approaches representing the system state as a net [3,4,5] are rather specific and have a limited range of application while the intent is to compare general-purpose object-oriented specification languages.

The paper is organized as follows. A pure algebraic object-oriented approach is presented in Section 2. Some applications of the set theory approach (*Object-Z* and *Z++* specification languages) and specification languages representing a

[*)] The research is supported in part by the Russian Foundation for Basic Research under the grant № 01-01-00787.

state as an algebra (*TROLL*, *Maude* and *Object-Oriented ASMs*) are described in Sections 3 and 4, respectively. Some conclusions and directions of further work are given in Section 5.

2 PURE ALGEBRAIC APPROACH

The approach of representing static aspects of a dynamic system in a formal algebraic framework (called a *pure algebraic approach*) has proved to be suitable for specifying important notions of the object-oriented paradigm, such as inheritance, subtyping and method overloading. This specification technique does not involve the notion of a state.

One of the most fundamental works on specifying classes and relations between them is done by F. Parisi-Presicce and A. Pierantonio. Late binding of object methods is formally introduced in [6]. These two works are presented below.

2.1 Relations between classes

The approach proposed in [7,8] is intended to give a strict algebraic definition of the notion of a class and to formalize the concepts of inheritance, actualization, combination, etc., which are considered as binary relations between classes.

The model of the class covers the features present in most object-oriented languages. First, it supports encapsulation as a distinction between what the class implements and what the designer of the class chooses to be visible. Then, two different roles of the class are distinguished: the class as a description of an object (instance of the class) and the class as a base for other classes. These roles result in different interfaces: an instance interface and a class interface, respectively (conventionally, the latter includes the former). For example, C++ *public* class members constitute an instance interface, while *protected* members extend it to the class interface. The class model also includes a parameter list, intended to model what is called *unconstrained* and *constrained genericity*, and an explicit *import interface*. It is important to note that the import interface specifies only what is needed, but not the class which is intended to provide the imported features. This allows different hierarchies to be built over classes though it is not a typical situation for object-oriented languages (it corresponds rather to the "include" interface than to direct inheritance). A special sort called a *class sort* models the concept of object identity*;* this sort is a distinguished "point" of a *pointed signature* and *pointed specification* introduced below.

In order to continue, some preliminary notes are required.

- A *signature* Σ is a pair (S, OP), where S is a set of sorts and OP is a set of function symbols. A pointed signature has a distinguished element $p(\Sigma) \in S$ that represents the sort of object identities.
- An *algebra* $A = (S_A, OP_A)$ of Σ is understood conventionally; the category of all Σ-algebras and morphisms between them is denoted by $Alg(\Sigma)$. For a sort name $s \in S$, the corresponding set of an algebra A is denoted by A_s; for an operation name $op \in OP$, the corresponding function in A is denoted by op_A.
- A *signature morphism* $h: \Sigma_1 \to \Sigma_2$ is a pair of functions $(h^S: S_1 \to S_2, h^{OP}: OP_1 \to OP_2)$, such that if $N: s_1 \dots s_n \to s \in OP_1$, then $h^{OP}(N): h^S(s_1) \dots h^S(s_n) \to h^S(s_n) \in OP_2$. A pointed signature morphism has an additional constraint: $h^S(p(\Sigma_1)) = p(\Sigma_2)$, i.e., the sort of object identities is mapped to the corresponding sort of object identities.
- Every signature morphism $h: \Sigma_1 \to \Sigma_2$, where $\Sigma_i = (S_i, OP_i)$, induces a *forgetful functor* V_h: $Alg(\Sigma_2) \to Alg(\Sigma_1)$. For a Σ_2-algebra $A' = (S_{2A'}, OP_{2A'})$, $V_h(A') = A = (S_{1A}, OP_{1A})$, where $S_{1A} = \{A'_h S_{(s)} \mid s \in S_1\}$, $OP_{1A} = \{h^{OP}(op)_{A'} \mid op \in OP_1\}$. That is, the family of sets S_{1A} consists of sets from $S_{2A'}$ so that, for each $s \in S_1$, the A'-set corresponding to $h^S(s)$is taken. The family of functions OP_{1A} consists of functions from $OP_{2A'}$ so that, for each $op \in OP_1$, the A'-function corresponding to $h^{OP}(op)$is taken. The result is an algebra since h is a signature morphism.
- The left adjoint of V_h is the *free functor* $Free_h: Alg(\Sigma_1) \to Alg(\Sigma_2)$; it makes Diagram 1 commutative for any pair of Σ_2-algebras A_2' and A_2'' and any morphism g between them; $u_h: ID_{Alg(\Sigma 2)} \to V_h \circ Free_h$ is called a *universal transformation.* For the proof of existence of u_h and $Free_h$ and further details refer to [9].
- An *algebraic specification* consists of a signature and a set of (positive conditional) equations E. The translation $f^{\#}(E)$ induced by the signature morphism f is understood conventionally (in each equation, all function names from OP_1 are substituted by corresponding function names from OP_2, and variables of the sort $s_1 \in S_1$ are treated as variables of the sort $f^S(s_1) \in S_2$).

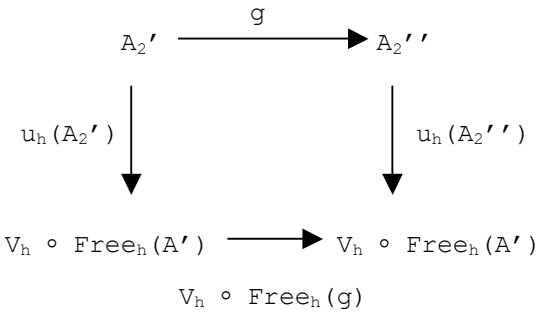

Diagram 1

– A *specification morphism* is a signature morphism, such that the translation $f^{\#}(E_1)$ is contained in E_2. For convenience, $p(\Sigma)$ is also denoted by $p(SPEC)$ for $SPEC = (\Sigma, E)$.

A *class specification* C_{spec} consists of five algebraic specifications PAR (parameter part), EXP_i (instance interface), EXP_c (class interface), IMP (import interface) and BOD (implementation part) and five specification morphisms as in the following commutative Diagram 2.

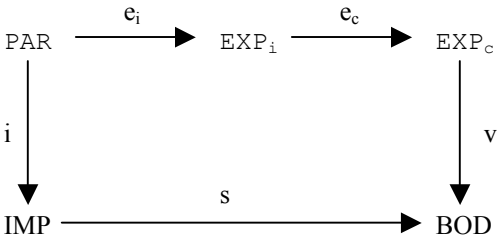

Diagram 2

The specifications EXP_i, EXP_c and BOD are pointed specifications, and e_c and v are pointed specification morphisms. *Semantics* $SEM(C_{spec})$ of the class specification C_{spec} is the composition $V_v \circ Free_s$: $Alg(IMP) \rightarrow Alg(EXP_c)$, that is a transformation from the models of the import interface to

the models of the export interface; it constructs a BOD-algebra from an IMP-algebra with all and only the features described in the body and then, forgetting all the hidden items, it returns an EXP_C–algebra. The class specification C_{spec} is *correct* if $V_s(Free_s(A)) = A$ for all $A \in Alg(IMP)$; this requires that operations (defined in the body) returning the values of imported sorts be total and well-defined.

A *class* $C = (C_{spec}, C_{impl})$ consists of a class specification C_{spec} and a class implementation C_{impl} such that $C_{impl} = Free_s(A_I)$ for some $A_I \in Alg(IMP)$. It requires a model of the import interface to be chosen; the class body can then be implemented.

This model allows formal definitions of both *reusing inheritance* and *specialization inheritance* as binary relations between classes. Suppose $C1 = (C1_{spec}, C1_{impl})$ and $C2 = (C2_{spec}, C2_{impl})$ are classes (algebras from $C1_{spec}$ are PAR_1, EXP_{i1}, EXP_{c1}, IMP_1 and BOD_1 and algebras from $C2_{spec}$ are respectively indexed by 2); then

- C2 *weakly reuses* C1 if there exists a morphism $f: EXP_{c1} \rightarrow BOD_2$ (exported features of C1 are present in the body of C2);
- C2 *strongly reuses* C1 if, in addition, $V_f(C2_{impl}) = V_{v1}(C1_{impl})$;
- C2 *is a weak specialization* of C1 if there exist morphisms
  $$f_i: EXP_{i1} \rightarrow EXP_{i2},$$
  $$f_c: EXP_{c1} \rightarrow EXP_{c2},$$
  such that $e_{c2} \circ f_i = f_c \circ e_{c1}$ (interfaces of C2 enhance those of C1);
- C2 *is a strong specialization* of C1 if, in addition, $V_{fc}(V_{v2}(C2_{impl})) = V_{v1}(C1_{impl})$.

In this way, specialization inheritance is a technique for defining the behavioral relation between classes, and it indicates that the instances of the new class (*subclass*) obey semantics of the *superlass*; each subclass instance is a special case of a superclass instance. Reuse inheritance, in contrast, is rather an implementation technique than a behavioral constraint: it indicates that everything in the class interface of C1 is available in the body part of the class C2 but not necessarily reexported by C2.

Two more relations correspond to instantiating the parameter part of a (generic) class and to replacing the import interface of a (semi-virtual) class with the export interface of another class; they are Wact (Sact in the strong form) and Wcomb (Scomb), respectively.

The above-presented approach has a solid mathematical background and covers static features of most object-oriented languages. However, it does not involve a notion of a state and state transformation and therefore is not suitable for specifying the evolution of a dynamic system. A special sort is introduced to model object identities, but this concept is not worked over in detail; the method overloading and late binding concepts are not considered at all.

2.2 Method overloading (static and dynamic)

One of the distinguishing features of the object-oriented approach is the fact that, in method calls, the actual method to be invoked is determined at run-time (*late binding*). In the languages like *C++* or *Java*, the correct method is selected basing on the *static type* of an object; this is defined in [6] as *dynamic overloading*, and it is one of the manifestations of *polymorphism*. The approach proposed in the paper deals with late binding in a formal algebraic framework, it also provides a capability of writing axioms related to a given type that are not required to hold for subtypes, thus reflecting at the specification level the fact that semantics of a method in a subtype may differ from that in the supertype.

The idea is to handle overloading at the *model* (semantic) level and not at the *signature* (syntactic) level. A function $op\colon \bar{s} \to s$ that has several variants is modeled by a *multifunction* that is a family of functions $op_{\bar{u}}$, one for each existing subtype $\bar{u}$ of $\bar{s}$. The main novelty of the logic part is the ability to distinguish between two different kinds of requirements (logical sentences) over elements of a given type `s`: those that must hold for elements of any possible subtype of `s` and those that hold for elements having `s` as the most specific type, but not for elements of its proper subtypes.

An order-sorted signature Σ is treated conventionally as a triple $(S,\leq,O)$, where $(S,\leq)$ is a preorder of sorts and O is a set of function signatures of the form $op\colon \bar{s} \to s$. There are three kinds of terms of Σ: *variables*, *function applications* and *casted terms*. They are built over the *S-indexed family of variables* $X = \{X_s\}_{s\in S}$ with pairwise disjoint subsets X_s. The S-indexed family of terms $T_\Sigma(X)$ is inductively defined by

- $X_s \subseteq T_\Sigma(X)_s$;
- $t \in T_\Sigma(X)_s$ and $s' \leq s$ implies $(s')t \in T_\Sigma(X)_{s'}$ (casting);

– if $t_i \in T_\Sigma(X)_{s''_i}$ and $s''_i \le s'_i \le s_i$ for all $i=1,\dots,n$ and *op*: $\bar{s} \to s$ is a function symbol, then $op(\tau_1,\dots,\tau_n) \in T_\Sigma(X)_s$, where $\tau_i = t_i$ or $\tau_i = t_i:s'_i$ for all $i=1,\dots,n$ (function application).

According to this definition, each term t has a unique type called its *static type*. But a term can be used as an argument in a function application whenever its static type is a subtype of the needed type; each supertype of the static type of t is called its *dynamic type*. In the above definition, the notation $t_i:s'_i$ means that t_i should be considered as a term of the sort s'_i.

An S-indexed family of logical sentences $P_\Sigma(X)$ is also defined inductively over terms from $T_\Sigma(X)$. *Atoms* over Σ and X are

– *definedness assertions* of the form $D(t)$, where $t \in T_\Sigma(X)$;
– *strong equalities* of the form $t = t'$, where the static type of t is s and the static type of t' is s' and there is a common supertype s^0 of s and s' ($s \le s^0$ and $s' \le s^0$), that is, there is a common dynamic type of t and t'.

Then, *horn clauses* from $HC(\Sigma,X)$ have the form $\varepsilon_1 \wedge \dots \wedge \varepsilon_n \supset \varepsilon_{n+1}$, where each ε_i is an atom. Finally, *conditional sentences* from $P_\Sigma(X)$ have the form $\forall x_1:s_{1\le} \dots \forall x_k:s_{k\le}.\forall x'_1:s'_1 \dots \forall x'_n:s'_n.\varphi$ for each $\varphi \in HC(\Sigma,X)$. The domains of x_i and x'_j are constructed differently (see below), so different notations are involved to define them ($\forall x_i:s_{i\le}$ and $\forall x'_j:s'_j$, respectively).

A Σ-model M is a family of sets $\{ s^M \mid s \in S \}$ and a family of functions (defined below). For each $s \in S$, a set s^M is called the *proper carrier* of the sort s in M. The *extended carrier* $s_\le^M$ of s in M is the union of all the proper carriers of all $s' \le s$:

$$s_\le^M = \{a: s' \mid a \in s'^M,\ s' \le s\}.$$

The notion of an extended carrier is expanded to sequences $\bar{s}$ of sorts with length n: $\bar{s}_\le^M = s_{1\le}^M \times \dots \times s_{n\le}^M$.

For each $op\colon \bar{s} \to s$ and each $\bar{u} \le \bar{s}$, M contains a partial function $op_{\bar{u}}^{M}\colon \bar{u}_{\le}^{M} \to s_{\le}^{M}$. The function is not necessarily total and it is essential due to non-termination of calculations in programming languages. Note that there is a family of functions associated with each operation signature.

Terms from $T_\Sigma(X)$ are interpreted (*evaluated*) in M basing on the valuation $V\colon X \to M$ of the family of variables:

- $x^{M,V} = V(x)$ for all $x \in X$;
- if $t \in T_\Sigma(X)_s$ and $s' \le s$ then $(s')t^{M,V} = t^{M,V}$, else $(s')t^{M,V}$ is undefined;
- $op(\bar{\tau})^{M,V} = op_{\bar{u}}^{M}(t_1^{M,V}, \ ..., \ t_n^{M,V})$, where u_i = $\begin{cases} s_i' & \text{if } \tau_i = t_i : s_i', \\ s_i'' & \text{if } \tau_i = t_i \text{ and } t_i^{M,V} = a_i \in s_i^{"M}. \end{cases}$

Satisfaction of logical sentences is defined conventionally following the intuition that a variable x_i can be evaluated to any value of the extended carrier of s_i, while x'_j has to be evaluated to the values of the proper carrier of s'_j.

According to the above model, the function to be invoked is determined basing on the static types of all function arguments; different functions are invoked for different sets of arguments (it is known as late binding in the OO terminology). Note that late binding of object methods is usually based on the static type of the object whose method is to be invoked, static types of the method arguments do not affect it. In this sense, the approach presented here is more general but it can be restricted to literally correspond to the OO concept of polymorphism.

This approach does not account for static overloading, but it can be safely extended. Static overloading can be resolved at compile time by modifying the signature Σ; more precisely, a distinct internal name is assigned to each overloaded function declaration and the resulting signature that differs from the original user-visible signature is used internally.

Finally, it is worth mentioning that the above approach does not use a notion of a system state and therefore is not suitable for specifying the system dynamics. The model of an object also seems to be quite rudimentary: although a special

sort (of object identities) is introduced for each object signature and there is a preorder of classes, the object identity is not used in invoking the methods of the class. This corresponds to the traditional procedural style of programming, rather than to the object-oriented methodology. As a conclusion, there is a formal introduction of late binding presented in the paper, but the object-oriented formal framework for it is incomplete.

3 SET-THEORY BASED APPROACH WITH STATES

The specification languages of this group formalize the dynamic system by constructing a mathematical model for it. Typically, a specification describes the states of a system (represented by data structures and relations between them) together with operations (formalized as functions) that transform one state into another. Semantics of these languages is based on the set theory. Due to its typical use — specification of states and operations — this group of languages seems to be well-suited for being adapted to the object-oriented paradigm.

One of the most widely used set-theory based specification languages is *Z* [10]. Several object-oriented extensions of Z have been developed; among them, we focus on *Object-Z* and *Z++*.

3.1 Object-Z

The specification language *Object-Z* is Z extended to facilitate specification in an object-oriented style. It came into existence in late 1988 as part of a collaborative project between the Department of Computer Science at the University of Queensland and the Overseas Telecommunications Corporation (OTC) of Australia. The goal of the project was to enhance structuring in the Z specification language (on which Object-Z is based) in order to more effectively specify medium-to-large-scale software systems. A more fundamental motivation was the desire to investigate the integration of formal techniques with the methodology of object orientation: a methodology which at that time was gaining rapid popularity in the programming community.

Now the language has reached a new level of completeness with the existence of axiomatic and denotational semantics and the first tools. Semantics of Object-Z is usually given on the basis of that of Z with the help of the metalanguage, such as the one introduced in [11], which allows the meaning of Object-Z constructs to be formally expressed in terms of constructs of Z.

Semantics of both Z and Object-Z are based on the set theory, and they use first-order logic to express system states and state transformations caused by an

operation execution. This is done by relating primed and non-primed variables that represent system states before and after the operation, as well as its input and output. In fact, Object-Z is a conservative extension of Z in the sense that all Z's syntax and its associated semantics are also a part of Object-Z. Therefore, any Z specification is also an Object-Z specification. On the other hand, Object-Z extends not only the syntax of Z but also the semantic universe in which specifications are given their meaning.

Syntactically, an Object-Z specification comprises a list of formal paragraphs — type definitions, axiomatic definitions, global predicates, schema definitions and class definitions — possibly interleaved with informal descriptive comments. Among these paragraphs, class definitions are of major concern since they are the only construct new to Z. Each class definition encapsulates a single state schema with its initial state schema and all the operations that can affect its variables. It syntactically differs from usual Z constructs since:

a) the roles of schemas within class definitions and their order are fixed;
b) the principle of "definition before use" does not hold for class operations and for class members whose values are object identities (this both facilitates writing recursive specifications and allows choosing a top-down design fashion).

A class definition comprises a named box possibly with generic parameters. This box may contain, in the order they can occur, a *visibility list* defining the class's interface, inherited class designators, local type and constant definitions, at most one *state schema*, the associated *initial state schema* and operation schemas.

The first construct in the class definition is the visibility list. It enumerates those features — constants, state variables, the initial state schema and operations — of the class that are in the class's interface and, hence, "visible" to the environment of objects of the class. The absence of the visibility list implies that all features are visible.

The inherited class designators comprise a class name, an instantiation of that class's generic parameters, if any, and possibly a rename list. This construct allows Object-Z classes to be directly reused in the definition of other classes. The type and constant definitions of the inherited classes and those declared explicitly in the derived class are merged. Any schemas with the same name and state schema are conjoined. Inheritance, therefore, is only possible when all names common to the inherited and inheriting classes are used for the same kind of definitions. Name clashes can be resolved by renaming.

The local type and constant definitions are as in Z, but with the scope limited to a single class. A state schema is nameless and comprises declarations of state variables and a state predicate. They are implicitly included in every operation and the initial state schema and form a class invariant. The initial schema that has a reserved name (namely `Init`) restricts the set of possible initial states; this name also can be used as a predicate to check whether an object is in its initial state.

The class operation schemas describe the methods defined for the class. They may use only the state variables of the object to which they belong, so the potential behavior of the class can be considered in isolation. A class operation schema extends the notion of a standard Z schema by adding a Δ-list to it. The Δ-list is a list of state variables that may be changed by an operation; all state variables not in the Δ-list remain unchanged. The pre- and postcondition describe the effect of an operation. Unlike Z, an operation cannot be executed if its precondition is violated (in Z, it can be executed but the outcome of the operation is undefined). All Z schema operations (conjunction, disjunction, etc.) can be used to form new operations as well as Object-Z specific operations: `[]` (non-deterministic choice), `||` (operator for inter-object communication), ⨟ (sequential composition). Operators `[]`, ∧ and ⨟ have a distributed form, which allows them to be applied to a collection of similar expressions. This facilitates specification of concurrent systems comprising variable-length collections of similar components that use aggregation, synchronization, communication and nondeterminism.

Semantic changes in Object-Z are consequence of support added for the *object identity* concept. It refers to that property of an object which enables it to be distinguished from all other objects. Support for it in Object-Z presents a major departure from semantics of Z. It allows the declaration of variables which, rather than directly representing a value, refer to a value in much the same way as pointers in a programming language. Semantics supporting such variables is called *reference semantics*.

Object identity is modeled in Object-Z by associating a set of values with each class. The sets for different classes are disjoint. For any class different from the one without objects (either because its initial state schema's predicate evaluates to false, or the class has no initial state schema and its state schema's predicate evaluates to false), this set is countably infinite. Each value associated with a class identifies a distinct object of the class. Object identities may be declared in a specification before the class of the referenced object is defined. This is possible since the set of object identities associated with the class A is independent of the actual definition of the class A.

An object variable can be declared polymorphically — that is, the identities of objects of different classes can be assigned to it (polymorphism in Object-Z is similar to genericity in the sense that it allows a variable to be declared which can be associated with more than one type). This can be done in two different ways. The first approach uses a syntactic construct of inheritance. The definition `a: ↓A` declares `a` as an object of the class `A` or of one of its subclasses (classes directly or indirectly inheriting `A` and having at least the visible properties of `A`). This means that the set of possible values is a union of all object identity sets for all A's subclasses. The second notion of polymorphism, called *class union*, is more flexible; it allows the declaration of an object of one of an arbitrary set of classes. Given the declaration `a: A∪B`, the identity of either `A` or `B` can be assigned to `a`. The associated set of values is a union of object identity sets for A and B, and possible operations over the variable are those belonging to the *polymorphic core* (defined as an intersection of the visibility lists of all the united classes). Despite its flexibility, the construct is hardly translated into most programming languages where polymorphism usually relies on inheritance. For polymorphically defined variables, the methods to be invoked are determined dynamically basing on the type of the associated object (late binding cannot be done for the usual class variables).

Object-Z has a specific notation to model *object containment* (the notion indicating that some object is contained within another one and cannot be referenced from other places external to the containing object). This notation enables the specifier to state that objects are contained in a system when they are declared. This is denoted by the subscript "$_{©}$" appended to the class name. Note that many objects still may refer to associated object in the specified system. However, it can be directly contained only in one object, say `a`, other objects should in their turn contain `a` in order to access the innermost containing entity, that is, they would access it indirectly. Object containment is sometimes possible to be modeled by conjoining properties and methods of the container and the contained entity and by considering the resulting subsystem as a single object of a new class, but this is not always the case (for instance, some objects of the inner class may exist independently and some may be contained in the outer object thus making conjoining of two classes impossible).

Creation and destruction of objects is not supported at the specification level. Each object that can be referenced by a specification exists throughout the evolution of the specified system — even in the case when it is never actually referenced. While this would not be feasible in a programming language where issues of memory usage and efficiency are important, it is not a problem in the abstract world of specification. This, however, complicates transforming a specification

into a concrete program and performing the related tasks automatically, but the problem is common for all set theory based formalisms: since the base set for each type (including object identities) is fixed, it cannot be populated with new elements during the program execution. This is the reason for selecting countably infinite object identity sets in Object-Z.

Another shortcoming of the approach is impossibility to consider a class as one of its superclasses and to call superclass operations for subclass objects. This is because object identity sets for such classes are disjoint and the "superclass-subclass" relation does not induce the inclusion relation between these sets.

3.2 Z++

Z++, as well as Object-Z, is among object extensions of the Z language. It arose from the Esprit II project REDO and the need within it to provide an abstract representation for large data-processing systems [12]. The aim was to produce a language that could naturally express design requirements, in addition to providing a means for structuring and incremental correct development of complex systems. In terms of the standard life cycle, it supports expression of requirements, design and specification. Support for implementation is now possible only via translation to a notation, such as B [13], which also allows the use of the semantic analysis facilities of the latter.

The concrete syntax of the language is quite different from that of Z and Object-Z. It was chosen in order to stress the commonalities of the concepts of the language with those of well-defined object-oriented languages such as Eiffel and, in addition, to simplify extensions of the notation by adding to the class definition new clauses rather than subordinate schema boxes.

The BNF description of Z++ class declaration is the following:

Object_class ::= `CLASS` ***Identifier TypeParameters***
[`EXTENDS` ***Imported***]
[`TYPES` ***Types***]
[`OWNS` ***Locals***]
[`RETURNS` ***Optypes***]
[`OPERATIONS` ***Optypes***]
[`INVARIANT` ***Predicate***]
[`ACTIONS` ***Acts***]
[`HISTORY` ***History***]
`END CLASS`

If the *TypeParameters* list is not empty, this means that a generic class is being defined (generic classes are templates for classes and, unlike non-generic classes, are not types themselves). Type checking over the selected generic parameters is possible: a type X in the list can be required to be a descendent of a class A via the notation A<<X, which means that A is an ancestor of X. In this way, constrained genericity is provided.

The EXTENDS list provides means for reusing other class definitions: it denotes the set of previously defined classes that are inherited in this class. The relation of inheritance generates two predicates:

$$\begin{aligned} &\mathit{inherit} \ll \\ &\mathit{tcinherit} \ \ <_i \end{aligned}$$

(the latter being the recursive extension of the former); they can be used across the specification. Some features of the inherited class may be hidden or renamed; this is useful for the purpose of reusability, separate refinement, and avoiding name clashes between the features upon extension.

TYPES are declarations of type identifiers used in the declarations of local variables of an object. The OWNS variable declarations are attribute declarations, in the style of variable declarations in Z. The INVARIANT gives a predicate that specifies the properties of the internal state, in terms of object attributes. This predicate is guaranteed to be true on the state of an object between executions of its operations. The default invariant (i.e., if this clause is omitted) is *true*.

The visible method declarations are divided in two groups. First of them is identified by the RETURNS list of operations as functions from a sequence of input domains to an output domain; these are operations with no side-effect on the state. The OPERATIONS list declares the types of all the remaining methods. The ACTIONS list gives actual definitions for all the operations that can be performed on instances of the object; the default action for a method, if no action for it is listed, is the completely non-deterministic operation on the state of the class and its parameter types. It is worth mentioning that the execution of an operation, in contrast to Object-Z, is possible even when its precondition is violated, but it may have an arbitrary behavior within the state and operation typing constraints in this case.

It is possible to mark some methods as 'spontaneous internal actions', using the symbol $*$ in front of their definitions. Such methods are not available to users

or inheritors of the class; they provide a means to model many real-world situations occurring spontaneously. One of such situations corresponds to *object sharing* that is possible for Z++ classes: when the same object is referenced more than once, changing its state by one of its "parents" looks like a spontaneous action for the others.

Two major features of the language semantics are strict mathematical concept of refinement and specific treatment of object identity.

Refinement (that is a synonym for subtyping and conformant subclassing in Z++) is the central concept of semantics; a strong mathematical definition is given for it. The refinement relation between classes C and D is expressed via the notation $C \sqsubseteq_{\phi,R} D$, where ϕ is a mapping of the method names of C to those of D, and R is a predicate which defines a data refinement relation between the attributes and constants of the class in the sense of Z. This reduces to $C \sqsubseteq_{\phi} D$, or $C \sqsubseteq_{R} D$ or $C \sqsubseteq D$ in the case when the specific interpretation function ϕ or the predicate R is of no interest ($C \sqsubseteq D$ is also a predicate that evaluates to *true* if there exist a refinement mapping and a relation from C to D). The notation $A \equiv B$ means that $A \sqsubseteq B$ and $B \sqsubseteq A$: such classes are semantically equivalent.

The existence of a refinement relation between two classes implies that the more refined class is type-compatible with the less refined class, in the usual sense of OO specification. It also allows some operations for composing classes, such as binary operation $\dot{*}$ producing the maximal common ancestor for two classes. The union operation, denoted as $\wedge$, creates a class that has the disjoint union of attributes of its arguments and conjoins the definitions of identically named methods. The operation $\cap$ forms the syntactic intersection of the definitions of two classes. Some operations over the generic class expressions, such as partial instantiation, are also available.

The concept of object identity is modeled by associating an infinite set @C (a set of *object identities* for potential and existing objects) with each class C. This set has a subset $\overline{C}$ of existing objects of C, and the dereference map $*_C$ is used to obtain object values from object identities of existing objects:

$$*_C : @C \rightarrow State_C$$
$$\overline{C} = dom\ (*_C)$$

($State_C$ is a state schema obtained by reducing the Z++ class declaration to the plain *Z*; it defines the states of all objects of C). This approach implicitly ad-

dresses the problem of memory management, since it allows distinguishing potential and existing objects of C, but it is not imperative and has nothing to do with real-life memory management issues.

If the declaration a: C occurs in the OWNS list of another class, this is interpreted as a: @C in the above translation. The class methods are also applied to object identities from @C, not to objects themselves; the method call syntax is a.m(e), where e is a list of arguments. There is a second use of methods as operations on the state of C. It requires the method m to be defined in the class D, a refinement of C, and the call syntax is either m(e) or C.m(e) if the former version is ambiguous. In this way, the OO concept of virtual and overloaded methods may be expressed in the specification.

The language is different from Object-Z in several key aspects. Z++ has no means of encapsulation like the visibility list of Object-Z: to hide private methods of the class, another class has to be defined. The language syntax also does not include the initial state schemas, though it may be easily extended. Z++ does not support polymorphyc declarations but it is equilibrated by existence of a strong refinement relation that induces type compatibility between more and less refined classes. It has more expressive parameterization including typed parameters and generic class expressions and supports not only declarative operation definitions (by relating pre- and post-conditions for the schema as in Z) but also the procedural style (using the B code). Object-Z defines several useful operations over objects and collections of objects, and Z++ allows operating over classes instead. Concerning mathematical foundations, both languages are well-defined by reductions to the plain Z notation. They also demonstrate problems common to all set theory based formalisms, such as poor support for object creation and destruction.

4 ALGEBRAIC APPROACH WITH STATES

The idea of representing the state as an algebra and state transformations as algebra transitions is rather new; several algebraic specification languages have been developed. Attractive properties of these languages (natural means for expressing most of programming language constructs, ease of specifying the state changes) result in numerous experiments in extending algebraic specifications towards object orientation. Object-oriented algebraic specification languages support the most important OO concepts (data encapsulation, inheritance, subtyping, object methods and method overloading, object creation and deletion). The most interesting of them (extensions of pure algebraic approach, *TROLL*,

Maude and *Object-Oriented TASMs*) are described below. We focus on several key features of these approaches:

- modeling the notions of object and object state;
- means of expressing object creation and deletion;
- support for various OO concepts.

4.1 Extensions of the pure algebraic approach

Though the class model introduced in ***2.1*** is well-tailored for describing the static aspects of a system (classes, their structure and relations between them), it does not address the system dynamics. The notions of a system state and state transformation are not defined; even an object as a class instance is informally introduced. Actually, this modification of a pure algebraic approach to the object-oriented specification is a tool for studying the static part of a dynamic system and cannot be regarded as a complete model of it. One of possible extensions of this approach that supports complex evolving object communities is formally introduced in [14].

The intent is to define an *object transition system (OTS)* in which every state is represented by an algebra that contains both objects and object values. The set of possible values is determined by the class implementation and it remains unchanged for all states (*instant algebras*); a *state function* associates with each object its current value. The OTS is formalized with the notion of category [9]. The objects of this category are instant algebras and morphisms model state evolutions. They occur whenever there is a *method execution* or *object creation/deletion*.

Given an algebraic class specification C_{spec}, the *object values specification* $SPEC^{\upsilon} = ((S,OP),E)$ of C_{spec} is obtained as the union of instance interfaces of C_{spec} and of all specifications used by C_{spec}. Its algebra A^{υ} is the *amalgamation sum* [9] of the corresponding implementation algebras; the algebra elements model all possible values of objects of C_{spec}. For each sort s in $SPEC^{\upsilon}$, two sorts are obtained: s^{υ} (the *sort of values* of the sort s), that is a renaming of s, and s^{Ω} (the *sort of objects* of s). Conventionally speaking, s^{Ω} models the set of object identifiers for objects of the class s. The authors designate $S^{\upsilon} = \{s^{\upsilon} \mid s \in S\}$, $S^{\Omega} = \{s^{\Omega} \mid s \in S\}$, both S^{υ} and S^{Ω} being signatures without operations. The set of state function signatures Ψ is defined as $\{\Psi^{\upsilon}{}_{s}: s^{\Omega} \rightarrow s^{\upsilon} \mid s^{\upsilon} \in S^{\upsilon}\}$. When interpreted, each state function defines the state of objects of the sort s (it maps object identities from s^{Ω} to object values from s^{υ}). The

state functions signature SIG^{Ψ} is defined in a natural way, as a pair $(S^{\upsilon} \cup S^{\Omega}, \Psi)$.

The category $Trans(A^{\upsilon})$ is the model where method executions are interpreted. Each *object* S of the category is a quadruple of algebras $(A3,A2,A0,A1)$, where

- $A0 \in Alg(S^{\upsilon})$,
- $A1 = A^{\upsilon} \in Alg(SPEC^{\upsilon})$ is an object value algebra,
- $A2 \in Alg(SIG^{\Psi})$,
- $A3 \in Alg(S^{\Omega})$,

with

- $V_{i1}(A1) = A0$ with $i1: S^{\upsilon} \to SPEC^{\upsilon}$,
- $V_{i2}(A2) = A0$ with $i2: S^{\upsilon} \to SIG^{\Psi}$,
- $V_{i3}(A2) = A3$ with $i3: S^{\Omega} \to SIG^{\Psi}$,

where V_{i1}, $V_{i2}(A2)$ and $V_{i3}(A2)$ are forgetful functors corresponding to signature morphisms $i1$, $i2$ and $i3$. The *canonical extension* $SPEC^{\Psi}$ of $SPEC^{\upsilon}$ is defined as follows. First, S is substituted by S^{υ} in $SPEC^{\upsilon}$; next, the specification is extended by S^{Ω} and Ψ. The state of the system is given by the algebra $A1 +_{A0} A2 \in Alg(SPEC^{\Psi})$ (this construct is a *pushout object* of $f_1: A0 \to A1$ and $f_2: A0 \to A2$; it is actually the disjoint union of $A1$ and $A2$, where $f_1(x)$ and $f_2(x)$ are identified for all $x \in A0$). Each *morphism* of $Trans(A^{\upsilon})$ is defined by a quadruple of mappings

$$(\Phi|_{S\Omega}: A3 \to A3', \Phi: A2 \to A2', id_{A0}:A0 \to A0, id_{A1}:A1 \to A1)$$

and a family of functions $\{f^{\circ}_s: A^{\upsilon}_s \to A^{\upsilon}_s \mid o \in A2^{\Omega}_s\}$, such that $\Psi^{A2'}_s(\Phi(o)) = f^{\circ}_s(\Psi^{A2}_s(o))$. For an object o of C_{spec}, the function f°_s from this family corresponds to the state change of the object due to the system transition identified by the morphism.

Certain morphisms in this interpretation correspond to method executions. Let $\omega \in A2^{\Omega}_s$ be an existing object of the sort s and $m: s_1,\ldots,s_n \to s$ be a method symbol in $SPEC^{\upsilon}$; execution of m over ω with parameters $a_1 \in A1^{\upsilon}_{s1}$, …, $a_n \in A1^{\upsilon}_{sn}$ causes the evolution described by the morphism $eval: S \to S'$ defined as $< (id_{A3}, id_{A2}, id_{A0}, id_{A1}), \{f^{\circ}_s \mid o \in A2^{\Omega}_s\} >$, where

$$f^{\circ}_{s} = \begin{cases} \lambda x.m^{A1}(x, a_1, \ldots, a_n) & \text{if } o = \omega, \\ id_{A1}{}^{\upsilon}{}_{s} & \text{otherwise.} \end{cases}$$

The identities say that the carrier sets remain the same, no object is created or deleted and only the object ω changes its current value (in general, the methods with side effects could be defined). Similarly, object creation and deletion are modeled.

4.2 TROLL

TROLL (*T*extual *R*epresentation of an *O*bject *L*ogic *L*anguage) is designed for the conceptual modeling and subsequent design of information systems [15,16]. The emphasis is on combining conceptual modeling and formal specification techniques with techniques for describing distribution and concurrency. As the main abstractions, TROLL supports *classes*, *roles* and *derived roles* (*specializations*), *composite objects*, *views*, and *relationships*. The basic concepts of TROLL can be characterized as follows:

- Objects are sequential processes which can synchronize their life-cycles through event interactions. The life of an object starts with a *birth* event and may end with a *death* event.
- Objects encapsulate an internal state that can only be modified by events. However, a part of the object state is observable through attributes.
- Objects are organized in classes; each object has a unique identity.
- An object is described by a set of attributes and events. Its evolution is specified by a possible sequence of events and communications with other objects.

Different formalisms are integrated into TROLL. The sorted first order logic is the basis for the state specification (the specification of data types is considered external to TROLL, and they are simply imported). The linear first order temporal logic (both past and future tenses) is used to describe the object behavior and state evolution constraints, respectively. The behavior of an object is defined as a linear process consisting of a set of possible traces of *event snapshots* (sets of concurrent events used to model object communication). A sublanguage

for the process specification is used to explicitly define the fragments of such life-cycles.

Syntactically, a TROLL specification is a set of templates of the following form:

```
template Name [ParameterList]
    <Imports and Declarations>
    <Component, Attribute, and Event Specification>
    <Object Behaviour Specification>
end template Name
```

A template may evaluate to an object class or may define behavior of individual objects not belonging to any class. Component, attribute, and event names must be unique in the template at hand, as well as the names of local classes.

Data type declarations can be built from imported data types, predefined types (`bool`, `nat`, `integer`, `real`, `char`, `string`) and data type constructors (`set`, `list`, `tuple`, and `enum`) and, together with local classes, can be used to define attributes. One special case is the object identity data types whose values are used as "handles" to objects. For a class C, its object identity data type is denoted as |C|; it can be used on a par with any other data type. With the specification of a class C, a function:

```
C: |C| → C-OBJECTS
```

is implicitly defined (C-OBJECTS denotes the set of all possible objects of the class C). The existence of such an artificial data type seems to be one of the most significant TROLL shortcomings: it means that semantics of the notion of an object is not sufficiently worked over. Conventionally, the algebraic semantics of the object includes a unique *identity* for each object and a set of *attribute function* values over this identity. The stand-alone identity adds an unnecessary level of indirection since it always stands for the object when an object is needed and this leads to an unnatural specification fashion.

The attribute and event specifications define the observable properties of objects and the state update operations, respectively. They form the local signature of an object. Attributes and events may be supplied with the named typed parameters. Data encapsulation is not directly supported: attributes and events are in the class interface unless they are `hidden`. A parameterized attribute defines one value for each combination of values of the parameter sorts (data types) — arrays can be specified in this way. Attribute values can be further restricted ei-

ther directly in the attribute definition after the `restricted` keyword or in the special `constraints` section of the template specification (in the latter case, dependences among attributes and dynamic constraints as Future Directed Temporal Logic (FDTL) predicates can be expressed). The attribute parameter values can also be restricted:

```
attributes
    SalaryInYear(Year:nat): money
        restricted Year > 1950 and Year <= 2010 .
```

In this example, for the years for which the formula evaluates to false, the value of the attribute denoted by SalaryInYear(Year) has to be undefined ("undefined" is a valid value of all data types in TROLL). By default, the attribute value is undefined when an object is created, but it may be explicitly changed in the `initialized` attribute definition section. `Derived` attributes are not stored; they are determined by a computation over the observable properties of an object instead.

Each event that can take place in the object's life is denoted by an event name and a list of (typed) formal parameters. Its definition can include the following sections: necessary conditions that should be fulfilled (`enabled`), local change of the state (`changing`), associations with other events (`calling`), and return values (`binding`). Spontaneous events are marked as `active`. Two special classes of events denote the creation (`birth`) and destruction (`death`) of objects. At least one birth event should exist for each template, but death events are optional.

Explicit pre-conditions for events, called *enabling conditions*, can be formulated in terms of predicate logic and Past Directed Temporal Logic (PDTL); alternatively, the *process description language* sketched below can be used. Such conditions are formulae built over attributes and event parameters. The special predicate *after* frequently used in this context is true in the state after the mentioned event took place. The state change is described by stating the value of the attribute after the occurrence of an action:

```
events
    deposit (amount : money)
        changing balance := balance + amount.
```

The term in the right-hand side is evaluated in the state before the event occurrence; the implicit *frame assumption* is that an attribute keeps its value if not explicitly changed. An optional condition formulated in terms of first order logic may precede each assignment in the section; the change is to be applied only if the condition is satisfied.

Calling rules of an event describe causal relationships to other events; each rule consists of a condition determining whether the call should be performed in the current state and of a list of event terms denoting the events to be called. For the event to take place, each of its dependent events should be permitted. If so, all of them are called together, that is, the transitive closure of the first triggered event defines the change of the state of the system. The construct is not usual for traditional object-oriented programming languages and is difficult for translation into any of them.

The *binding rules* are special means of describing the output parameters of an event. It should be noted that such output parameters cannot participate in enabling conditions or calling conditions of the event since they are initially undefined.

The *process language* introduced in TROLL provides mechanisms to describe possible evolution of objects in terms of sequences of events. Each lifecycle description is divided into the declaration part and usage part. Processes may be combined with several operators such as *sequencing*, *choice*, *parallel,* and *for each*; recursive processes are allowed. The `start` condition is a PDTL formula that denotes the state of an object where the process becomes valid (may be initiated). The `interleaving` mode specification serves to describe various possibilities for other events to interleave the specified sequence (by default, such events are possible but their names should not be mentioned in the process declaration; other options are *none*, *free* and *excluding* modes). Declaring a process as `initiative` means that all its events become *active* (spontaneous) as long as the process is executed.

As mentioned above, templates evaluate to classes and/or to individual objects. There is a distinction between templates and classes: a template is a static description of the object structure and behavior. Classes, however, are dynamic since there is a time varying set of objects in them.

Objects may be *components* of other objects, which are called *composite objects* in this case. The relation between an object and its component is stronger than so-called object referencing, since it is bi-directional (components may use properties of the composite object). In TROLL, the composition may change dynamically; *set-valued* and *list-valued* components are allowed (the relation between a set-valued component and its elements is the same as the one between

a class and its objects). Components may interact with objects containing them and with each other by means of *interaction rules*.

Different forms of object `identification` are allowed, such as specification of unique *key attributes* of a class to identify objects or direct identification by name for single objects. The empty `identification` section of a class means that its objects are *anonymous*; they should be identified by means of object identity values of the class.

Objects are related in various ways. A `role` describes a temporal specialization of an object having additional properties and/or restricted behavior. A *specialization* is considered to be a permanent role. Over the objects, `views` can be defined which are means for the description of a restricted access to the set of objects of the class (a *selection view*) or to the set of properties of objects (a *projection view*).

To describe communication between separate objects, a special kind of global constraints called `relationship` can be used. They either relate the states of different objects or define causality between events of different objects. Relationship rules can be used to model the data-flow between related objects via binding of event parameters.

TROLL has a special means to operationally express object creation, management and destruction. A *class object* is implicitly generated for every class; conceptually it is not different from any other object. For a class `C`, a class object has a single set-valued component called `Objs` that represents all existing objects of `C`, a key map (a set-valued attribute of the type `|C|`) for each key attribute of `C`, several derived attributes, events and constraints to manipulate objects of `C`.

There is a simplified version of the TROLL called *TROLL light* that is tailored towards verification and direct execution purposes. It does not support object sharing, recursive structures and conditions formulated in the temporal logic; the relationships between the states and events of different objects, as well as object creation and destruction, cannot be expressed. The semantics of TROLL is mostly informal; in contrast, the algebraic semantics of TROLL light is given in [17]. The underlying model of an object community is an algebraic transition system, where both the state and the state transformation caused by finite sets of events are represented as algebras.

A single data algebra `DA` of the signature `DΣ = (DS,Ω)` is assumed to be fixed, where `DS` is a set of data sorts and Ω is a family of sets of operation symbols. The attribute signature `AΣ` extends the data signature with the set of object sort symbols `OS` and the set of attribute symbols `A`, so that `AΣ = (DS ∪`

$OS, \Omega \cup A)$. Algebras of this signature represent the system states (they are so-called *instance algebras*). The event signature extends the attribute signature, so that $E\Sigma = (DS \cup OS \cup \{\hat{e}\}, \Omega \cup A \cup \hat{E})$, where $\hat{e}$ is a special event sort and $\hat{E}$ is a set of event symbols. Algebras of $E\Sigma$ (*event algebras*) describe possible state changes caused by the set of events. A single state transition is represented by a triple $<A_L, \hat{A}, A_R>$ (the instance algebras A_L and A_R before and after the change are joined with the event algebra $\hat{A}$), and an object community $M_{O\Sigma}$ is a subset of $ALG_{A\Sigma} \times ALG_{E\Sigma} \times ALG_{A\Sigma}$. Various syntactic constructs of the language become axioms restricting the set of possible triples. State constraints and derivation rules affect A_L and A_R; calling rules influence A_L and $\hat{A}$; changing rules of events and behavior definitions control A_L, $\hat{A}$ and A_R in the whole.

The latest (third) version of TROLL [18] is different from the above-described TROLL 2.0. It is devoted to modeling the distribution issues and has been designed with the aim of executability. The language now combines graphical (*OmTROLL*) and textual (*TROLL*) notations for system specification. The graphical notation serves for description of global structures and their correlations; the textual notation is used for description of local properties and for detailed description of global relationships and constraints. The language allows more expressive data type definitions, component definitions and system behavior specifications.

4.3 Maude

Maude [19,20] is an object-oriented executable specification language and a system supporting both *equational* and *rewriting logic* concurrent computation for a wide range of applications, including automated proof systems, cryptographic protocols and network applications. Maude uses and extends the algebraic specification paradigm, and its equational logic sublanguage essentially includes a well-known specification language OBJ3 [21].

The language consists of the basic part (Core Maude) and its reflection called Full Maude. Core Maude supports module hierarchies consisting of (non-parameterized) *functional* and *system modules* and provides the META-LEVEL module that allows operating in the *module algebra*, where *modules*, *terms* and *kinds* themselves act as terms. Full Maude is an extension of Core Maude, written in Core Maude itself, that supports the module algebra of *parameterized modules*, *views*, and *module expressions,* as well as *object-oriented modules* with a convenient syntax for object-oriented applications.

Though Core Maude is not object-oriented, its components (functional and system modules) are basic specification blocks and should be discussed first. The

functional modules define the equational theories whose equations are Church-Rosser and terminating. A mathematical model of data and functions is provided by the initial algebra defined by the theory whose elements are equivalence classes of ground terms modulo equations. The task of the Maude interpreter is to reduce a given term to its ground (canonical) form using a given theory. The equational logic on which Maude functional modules are based is an extension of the order-sorted equational logic called *membership equational logic*; it supports so-called *membership axioms*, in which a term is asserted to have a certain sort if a condition is satisfied.

In the functional module, *sorts* (viewed as algebraic data types) and *subsort relations* (inducing set-theoretic inclusions between the corresponding sets of the algebra) are defined by sort declarations and subsort declarations. Each equation may have previously declared variables of appropriate sorts. An *operator* can be declared together with its properties (like associativity) and evaluation strategy. An *error sort* is added to the top of each chain of sorts partially ordered by the subsort relations; such error sorts are called *kinds.* A term is assumed to belong to an error sort if it does not satisfy any of the membership axioms for other sorts; in this way, the Maude interpreter reports errors.

Each system module specifies the initial model of a *rewrite theory*. The theory essentially contains a theory in the membership equational logic, but terms in the rewrite theory are no longer interpreted as functional expressions — they now represent the states of the system. A set of possibly conditional labeled *rewrite rules* defines state transitions. These rules need not be Church-Rosser and terminating; consequently, many reactive systems so specified should never terminate, and a system may evolve in a highly nondeterministic way. Therefore, the issue of executing specifications for system modules is considerably subtler than executing expressions in a functional module. The Maude interpreter provides a *default strategy* for executing expressions in system modules; it applies the rules in a top-down fair way. Other strategies can be specified within Maude in the user-definable extensions of the module META-LEVEL. This technique makes use of the fact that rewriting logic is reflective, i.e. it can be faithfully interpreted in itself.

Besides the strategies, reflection makes possible many advanced metaprogramming applications. One of such applications is Full Maude, which makes essential use of reflection to provide Maude with a rich and extensible module algebra. In Full Maude, concurrent object-oriented systems can be defined by means of object-oriented modules using syntactic constructs, such as classes, objects, messages and configurations, which are more convenient than those of system modules. A class definition `class C | att`$_1$`: s`$_1$ `...` `att`$_n$`: s`$_n$

defines a class `C` with the attribute names $\texttt{att}_1, \ldots, \texttt{att}_n$ and their corresponding sorts (classes without attributes can be defined using the syntax `class C` though the rationale of having such classes is not clear). Then objects are record-like structures of the form $\texttt{<O: C | att}_1\texttt{: v}_1\ \texttt{...}\ \texttt{att}_n\texttt{: v}_n\texttt{>}$, where `O` is a user-defined name serving as object identity and the $\texttt{v}_i$'s are the corresponding attribute values. An object without attributes can be represented as `<O: C |>`. Objects can interact with each other by means of messages. A message declaration `msg m: p1 ... pn` → `Message` (where `Message` denotes a single pre-defined sort for modeling class messages) defines the name of a message and the parameter sorts; the message itself is then denoted as $\texttt{m(v}_1\texttt{, …, v}_n\texttt{)}$. The first parameter often refers to the message recipient. The state of a concurrent object system, that is a multiset of objects and messages, is called a *configuration*. The dynamic behavior of a system is axiomatized by specifying each of its concurrent transition patterns by a corresponding rewrite rule: either `rl[label]: C` ⇒ `C'` or `crl[label]: C` ⇒ `C' if cond` for conditional transactions. Both `C` and `C'` are subconfigurations (subsets of the configuration; there is no need to mention objects and messages not affected by the transition). In general, `C` may include several objects describing synchronization or multi-party interaction between them; object creation and destruction can be expressed as well. Full Maude supports constrained genericity for object modules; generic module instantiation is considered to be a special operation in the module algebra on the meta-level.

Maude's order-sorted type structure provides a natural support for both single and multiple class inheritance. A subclass declaration `C < C'` in an object-oriented module is a particular case of a subsort declaration. Its effect is that the attributes, messages, and rules of all the superclasses, as well as the newly defined attributes, messages, and rules, characterize the structure and behavior of the objects in the subclass (it behaves exactly as any object in any of the superclasses, but it may exhibit additional behavior due to the introduction of new properties). Name conflicts are not resolved (implicit renaming is not supported); this results in additional constraints on inherited classes; alternatively, explicit renaming is possible to be expressed by means of `views` as signature morphisms.

Maude does not support data encapsulation; each and every data field of the class is visible for its users and descendants. Furthermore, class methods are not supported in any way (messages are not bound to any particular object and therefore cannot be regarded as their alternative). These shortcomings would impede using Maude as a tool for object-oriented software design.

Although object-oriented modules facilitate programming of concurrent object systems, each such module can be reduced to the corresponding system module (a special object-oriented syntax can be regarded as syntactic sugar). This transformation introduces a single global superclass `Cid` for all classes. A subsort S_C of a sort S_{Cid} is introduced for each class `C`; elements of S_C are object identities for objects of `C`. The class attributes are mapped to functions taking object identity as an argument. The subclass declarations result in the subsort declarations; rewrite rules are modified to make them applicable to all objects of the given classes and their subclasses. Thus, semantics of an original object-oriented module is that of the corresponding system module.

Each system module defines a theory Ψ in the rewriting logic that is a 4-tuple $(\Sigma; E; L; R)$, where Σ is a ranked signature of function symbols, `E` is a set of Σ-equations that are Church-Rosser and terminating (they define equivalence classes for terms of Σ), `L` is a set of labels, and `R` is a set of pairs belonging to $L \times T_{\Sigma;E}(X)^2$, whose first component is a label and the second component is a pair of `E`-equivalence classes of terms with $X = \{x_1; \ldots; x_n; \ldots\}$ being a countably infinite set of variables. Elements of `R` are called *rewrite rules*; a rule $(r, ([t], [t']))$ is a transition $r: [t] \rightarrow [t']$, where `[t]` and `[t']` are `E`-equivalence classes identified by their representative terms `t` and `t'`. A rewrite theory Ψ entails a sentence $[t] \rightarrow [t']$ if and only if it can be obtained by finite application of deduction rules (reflexivity, congruence, replacement and transitivity) in this framework.

The model of a rewrite theory Ψ is the category $T_\Psi(X)$ whose objects are equivalence classes of terms $[t] \in T_{\Sigma;E}(X)$ and whose morphisms are equivalence classes of "*proof terms*" representing proofs in rewriting deduction. The rules for generating such terms, with the specification of their respective domains and codomains, just "decorate" the rules of deduction in rewrite logic; for instance, the composition rule decorates transitivity:

$$\frac{\alpha : [t_1] \rightarrow [t_2] \quad \beta : [t_2] \rightarrow [t_3]}{\alpha;\beta : [t_1] \rightarrow [t_3]}.$$

Proof terms form an algebraic structure $P_\Psi(X)$ consisting of a graph with nodes $T_{\Sigma;E}(X)$, with identity arrows, and with operations `f` (for each $f \in \Sigma$), `r` (for each rewrite rule), and "`;`" (for composing arrows). The model $T_\Psi(X)$ is the quotient of $P_\Psi(X)$ modulo the following equations:

1. *Category.*

(a) Associativity. For all α, β, γ: $\alpha;(\beta;\gamma) = (\alpha;\beta);\gamma$.

(b) Identities. For each α: $[t] \rightarrow [t']$:

$$[t];\alpha = \alpha;[t'] = \alpha.$$

2. *Functoriality of the algebraic structure.* For each $f \in \Sigma_n$, the following holds:

(a) Preservation of composition. For all $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$:

$$f(\alpha_1;\beta_1, \ldots, \alpha_n;\beta_n) = f(\alpha_1, \ldots, \alpha_n);f(\beta_1, \ldots, \beta n).$$

(b) Preservation of identities.

$$f([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)].$$

3. *Axioms in* E. For the axiom $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ in E and for all $\alpha_1, \ldots, \alpha_n$:

$$t(\alpha_1, \ldots, \alpha_n) = t'(\alpha_1, \ldots, \alpha_n).$$

4. *Exchange*. For each $r: [t(x_1, \ldots, x_n)] \rightarrow [t'(x_1, \ldots, x_n)]$ in R:

$$\frac{\alpha_1 : [\varpi_1] \rightarrow [\varpi'_1] \quad \ldots \quad \alpha_n : [\varpi_n] \rightarrow [\varpi'_n]}{r(\overline{\alpha}) = r([\overline{\varpi}]); t'(\alpha) = t(\alpha); r([\overline{\varpi}'])}.$$

The first group of equations makes $T_\Psi(X)$ a category; the second one makes each $f \in \Sigma$ a functor, and the third group forces the axioms E to be satisfied in the model. The exchange equation states that rewrite rules can be applied in any order, sequentially or simultaneously.

4.4 Object-Oriented Abstract State Machines

The approach presented in [22] uses the mechanism of Typed Abstract State Machines (TASMs) [23] to provide means of object-oriented specification of complex evolving systems. The specification of such a system includes specifications of *data types* representing the static part of a system, *object types* representing the dynamic part of a system and independent *functions* and *procedures* (they are not included into this review for simplicity). Each state of a system is modeled by a many-sorted algebra (*instance algebra*); state transitions are defined by *transition rules* of the TASM.

An object is understood as a complex entity possessing a unique *identifier* and *state*, that is a set of *attributes*. The object state can be initialized, updated and viewed; object *methods* are accordingly divided into *initializers*, *mutators* and *observers*. An object can belong to several object types in compliance with the *supertype/subtype* hierarchy; this relation also provides the base for *late binding* of object methods.

A *type-structured signature* Σ_{dat} is a pair $<TYPE, \Omega>$, where TYPE is a set of data type names and Ω is a set of function signatures over TYPE (each function signature has the form $op: T_1, \ldots, T_n \rightarrow T$, where op is a function name and each of T and T_i, $i=1, \ldots, n$, is an element of TYPE). A *data type specification* is a data type signature with a set of axioms E. An algebra of such a specification is built conventionally by associating the sets of elements with the names from TYPE (for the name $T \in TYPE$, we denote the corresponding set in the algebra A by A_T) and partial functions with the function signatures from Ω so that each axiom from E is satisfied. Such an algebra is called a *static algebra* in the sequel.

A *dynamic system signature* $D\Sigma$ extends Σ_{dat} in the following way: let OTYPE be a set of *object type names* such that $OTYPE \cap TYPE = \varnothing$ and

- an *observer profile* be either T or $T_1, \ldots, T_n \rightarrow T$,
- a *transformer profile* be $T_1, \ldots, T_n$,

where $T, T_i \in TYPE \cup OTYPE$, $i=1, \ldots, n$. Then

- an *attribute signature* is a pair $at: OP$, where at is a name and OP is an observer profile (attribute profiles of the form $T_1, \ldots, T_n \rightarrow T$ serve for modeling multidimensional arrays of elements of T);
- an *observer signature* is a pair $b: OP$, where b is a name and OP is an observer profile;
- a *transformer signature* is either m or $m: MP$, where m is a name and MP is a transformer profile; transformers are divided into *initializers* and *mutators* because of slightly different roles they play;
- an *object type signature* is a set of the attribute, observer and transformer signatures.

The signature $D\Sigma$ is then defined as a pair $<\Sigma_{dat}, \Sigma_{obj}>$, where $\Sigma_{obj} = <OTYPE, O\Phi, int^\circ>$, $O\Phi$ is a set of object type signatures, int° is a function

that maps the object type names into $O\Phi$. For the name T mapped into the signature ϕ, we say that ϕ is marked with T. The sets of the attribute, observer and mutator signatures in the object type signature marked with T are denoted as $Att(T)$, $Obs(T)$ and $Mut(T)$, respectively.

An object type T_1 is a *subtype* of an object type T (denoted by $T_1 < T$) if $Att(T) \subseteq Att(T_1)$ and $Obs(T) \subseteq Obs(T_1)$ and $Mut(T) \subseteq Mut(T_1)$. An object type T is a *root type* for an attribute $at: OP \in Att(T)$ (at is, respectively, a *root attribute* of T) if there is no T_1 such that $T < T_1$ and $at: OP \in Att(T_1)$.

An *instance algebra* A of the dynamic system signature $D\Sigma$ is a typical order-sorted algebra built as an extension of a static algebra B of Σ_{dat} in the following way:

- with each object name $T \in OTYPE$, a set of elements A_T is associated reflecting the supertype/subtype hierarchy: if $T < T'$, then $A_T \subset A_{T'}$ (these elements are called *object identifiers*);
- with each root attribute $at: T_1, \ldots, T_n \rightarrow T'$ in the object type signature marked with T, a partial function $at^A_T: A_T \rightarrow (A_{T1}, \ldots, A_{Tn} \rightarrow A_{T'})$ called an *attribute function* is associated. A non-root attribute $at: OP$ in the object type signature marked with T_1, such that $T_1 < T$ and T is a root type of at, is mapped to the same function as in T.

An *object-oriented dynamic system* $D(B)$ of the signature $D\Sigma$ consists of a set OID of object identifiers and a set $|D(B)|$ of instance algebras of $D\Sigma$ having the same static algebra B such that for each $A \in |D(B)|$ and for each $T, T' \in OTYPE$

- $A_T \subset OID$ (object identifiers are always chosen from the same set, namely OID);
- $A_T \cap A_{T'} = \varnothing$ if neither $T < T'$ nor $T' < T$ (the sets of object identifiers for different types are disjoint).

It also includes a set of *algebra modifiers* and a set of observers and transformers.

Algebra modifiers transform one instance algebra into another. Two kinds of modifiers (μ_1 and μ_2) change partial function interpretation at a certain point; the other two (μ_3 and μ_4) serve for creation and deletion of objects. Informally

speaking, μ_1 redefines a function at a given point, μ_2 makes it undefined at that point, μ_3 expands the set of object identifiers of a certain object type and all its supertypes by a new identifier, and μ_4 deletes one object identifier of a certain object type from the current state and changes function interpretations accordingly. An algebra modifier applied to the instance algebra produces an *algebra update*; several modifiers applied simultaneously produce an *update set*. The set of all *update sets* in D(B) is denoted by Γ. More technical details can be found in [22].

Given a dynamic system D(B), observers and transformers are interpreted by associating

- with each observer signature $b: T_1,\ldots,T_n \to T'$ in the object type signature marked with T, a partial map $b^{D(B)}{}_T: |D(B)| \to (A_T, A_{T1},\ldots,A_{Tn} \to A_{T'})$ called an *observer*;
- with each transformer signature $m: T_1,\ldots,T_n$ in the object type signature marked with T, a partial map $m^{D(B)}{}_T: |D(B)| \to (A_T, A_{T1},\ldots,A_{Tn} \to \Gamma)$ called an *initializer* or *mutator*.

Note that if $T < T'$, different maps $f^{D(B)}{}_T$ and $f^{D(B)}{}_{T'}$ can be built in D(B) to interpret the same observer or transformer name f present in both T and T′; this corresponds to the principle of *dynamic overloading* typical of object-oriented programming paradigm.

Given a dynamic system signature $D\Sigma=<\Sigma_{dat},\Sigma_{obj}>$, the set of $D\Sigma$-*terms* is defined as an extension of the set of Σ_{dat}-terms. Terms are built with the use of attribute, observer and transformer names (such terms are called *attribute values*, *observer* and *transformer calls*, respectively). The terms are interpreted with the use of above-defined attributes, observers and transformers. When a term interpretation is an update set, it is called a *transition term* (such terms denote transitions from one algebra to another). With the exception of transition terms, each term t is of type $T \in TYPE \cup OTYPE$ (each supertype T_1 of T is also the type of t since interpretation $t^A \in A_T$ of t is also an element of A_{T1}). If $t^A \in A_T$ and there is no $T'<T$ such that $t^A \in A_{T'}$, then T is called the *static type* of t; each supertype T_1 of T is the *dynamic type* of t. Observer and transformer calls of an object identified by a term t are interpreted using the static type of t; this is known as *late binding*. A special transition term, new(y: T) in τ, where $T \in OTYPE$, y is a variable of type T, and τ is a transition term using y, provides a means of object creation and its further use in τ. It is interpreted as an

algebra update that adds a new element to A_T, assigns it to y and performs the algebra update τ to initialize and use the newly created object. Other means of defining transition terms are called *transition rules*. The *basic transition rules* redefine attribute functions at certain points; *rule constructors* such as a *sequence constructor*, *set constructor*, *guarded update* and *loop constructor* serve for recursive constructing of complex transition terms from the simpler ones.

If t_1 and t_2 are two $D\Sigma$-terms of type T, then $t_1 == t_2$ is a *static equation*. It holds in an instance algebra A if $t_1^A = t_2^A$. If t_1 and t_2 are two transition terms of $D\Sigma$, then $t_1 == t_2$ is a *dynamic equation*. It holds in D(B), if for any $A \in D(B)$ 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 produces the same algebra B. A *dynamic system specification* over an object-structured signature $D\Sigma$ is a pair $<D\Sigma, E>$, where E is a set of equations that contains:

- a subset SE_{dat} such that $<\Sigma_{dat}, SE_{dat}>$ is a data type specification;
- a subset $E^{\circ\phi}$ for each object type signature $\circ\phi$ from $D\Sigma$. $E^{\circ\phi}$ contains a set of static equations for each observer name in $\circ\phi$ and a set of dynamic equations for each transformer name in $\circ\phi$. A pair $<\circ\phi, E^{\circ\phi}>$ is called an *object type specification*.

It is worth mentioning that the resulting specification language has some important advantages:

- a transition term looks like an imperative statement, it is easy to understand and it is executable;
- the data type specification level is separated from the object type specification level; similar means of specifying both levels are provided;
- the most important features of the object-oriented paradigm (inheritance, polymorphism, subtyping) are fully supported.

The above approach can be extended to provide means of specifying the constrained generic object types [24].

5 CONCLUSIONS

In this review, the main trends in object-oriented dynamic system specification have been discussed. Now we briefly summarize the results of the discussion.

Pure algebraic specifications are suitable for modeling the static aspects of a system, including class hierarchies, genericity and methods overloading. However, since no notion of state is formally introduced, it is impossible to specify the dynamic aspects (behavior of a system) in this framework.

Set-theory based specifications allow defining both static and dynamic system properties since they have a notion of state. Two object-oriented extensions of Z discussed above exhibit nearly equal expressive power; they support the main OO concepts and allow executable specifications to be defined. The most important shortcoming of both *Object-Z* and *Z++* is poor support for object creation and destruction — they cannot be expressed naturally since adding/removing elements to the base sets is impossible.

Algebraic specification formalisms with state provide better support of dynamic operations over objects, including their construction/destruction. In comparing these approaches, we focus on semantic details: how an object and object state are modeled, what OO concepts are supported. One of attractive properties of the language is *executability* of specifications: on the specification stage, it is often desirable to have a working model of a dynamic system.

The extension of the pure algebraic approach presented above certainly has a solid mathematical foundation. A support of object orientation, in contrast, is rather limited (no subtyping, inheritance, overloading). The model of the object seems to be unnatural (two sets for each object type are involved); the specification is hard to understand and it is not executable.

TROLL supports object methods, subtyping and genericity. Data encapsulation is not directly supported; method overloading is not possible. The notion of object is not sufficiently worked over: the object identity data type is superfluous and adds an unneeded complexity. *TROLL* itself is not executable specification language but *TROLL light* is.

Maude is a powerful framework for complex concurrent computations that allows choosing a computation strategy. Support for metaprogramming applications and strong mathematical foundations make it a perfect automatic proof system. However, object orientation in *Maude* is quite rudimentary: objects are just tuples of values, the language does not support data encapsulation and overloading; the notion of inheritance is not different from subtyping.

Finally, *Object-Oriented ASMs* has been considered. Benefits of this specification approach are:

- natural semantics for the notions of object and object state;
- support for object creation and deletion;
- support for data encapsulation, polymorphism, subtyping, overloading, constant and mutable objects;
- typical programming language constructs (conditional statements, loops, etc.) are modeled as complex algebra transitions; as a result, an *OO-ASM* specification is easy to understand and it is executable.

Though the specification facilities are sufficiently powerful, a support for several OO concepts (inheritance and genericity are most important among them) is incomplete and needs to be worked over in detail in a concrete language. Another important task is to create an interpreter for *OO-ASMs*; this would make the language practically useful on the early stages of complex software engineering and validation. These are subjects of further investigation.

REFERENCES

1. **Felder M., Morzenti A**. A Temporal Logic Approach to Implementation and Refinement in Timed Petri Nets // Lect. Notes Comput. Sci. — 1994. — Vol. 827. — P. 365—381.
2. **Morzenti A., San Pietro P.** An Object-Oriented Logic Language for Modular System Specifications. — Milan, 1990. — (Tech. Rep. / Dipartimento di Elettronica, Politecnico di Milano; No 90-27).
3. **Battistion E., Chizzoni A., De Cindio F.** Modeling a Cooperative Environment with Clown // Proc. of the 2nd Intern. Workshop on "Object-Oriented Programming and Models of Concurrency" within the 16th Intern. Conf. on Application and Theory of Petri Nets. — Osaka, Japan, 1996. — P. 12—24.
4. **Sibertin-Blanc C.** Cooperative Nets // Lect. Notes Comput. Sci. — 1994. — Vol. 815. — P. 471—490.
5. **Lakos C. A.** Loopn++ User Manual. — Hobart, 1996. — (Tech. Rep. / Dept. Comput. Sci., Univ. Tasmania; No R96-1).
6. **Ancona D., Cerioli M., Zucca E.** A Formal Framework with Late Binding // Lect. Notes Comput. Sci. — 1999. — Vol. 1577. — P. 30—44.
7. **Parisi-Presicce F., Pierantonio A.** Structured Inheritance for Algebraic Class Specifications // Lect. Notes Comput. Sci. — 1994. — Vol. 785. — P. 295—309.

8. **Parisi-Presicce F., Pierantonio A.** Reusing Object-Oriented Design: An Algebraic Approach // Lect. Notes Comput. Sci. — 1994. — Vol. 858. — P. 329—345.
9. **Ehrig H., Mahr B.** Fundamentals of Algebraic Specification 1. Equations and Initial Semantics. — Springer-Verlag, 1985. — 325 p.
10. **Spivey J. M.** The Z Notation. — Prentice Hall, 1992. — 150 p.
11. **Smith G.** The Object-Z Specification Language. — Kluwer Academic Publishers, 2000. — 146 p.
12. **Lano K., Haughton H.** The Z++ Manual. — Waddon, 1994. — (Tech. Rep. / Dept. of Computing, Imperial College).
13. **Abrial J.R.** The B Book — Assigning Programs to Meanings. — Cambridge University Press, 1996. — 850 p.
14. **Pierantonio A.** Making Statics Dynamic: Towards an Axiomatization for Dynamic ADTs // Proc. Intern. Workshop "Quality of Communication-Based Systems" — Berlin, 1994. — P. 19—34.
15. **Hartel P., Hartmann T., Kursch J., Saake G.** Specifying Information System Dynamics in TROLL // Proc. Workshop "Formal Methods for Information System Dynamics". — Univ. of Twente, 1994. — P. 53—64.
16. **Hartmann T., Saake G., Jungclaus R., Hartel P., Kursch J.** Revised Version of the Modelling Language TROLL (Version 2.0). — Braunschweig, 1994. — (Tech. Rep. / Technische Universität Braunschweig; No 94-03).
17. **Gogolla M., Hertzig R.** An Algebraic Semantics for the Object Specification Language TROLL light // Lect. Notes Comput. Sci. — 1995. — Vol. 906. — P. 288—304.
18. `http://www.cs.tu-bs.de/idb/publications/tr97/tr97.html` — TROLL 3 tutorial.
19. **Denker G., Meseguer J., Talcott C.** Protocol Specification and Analysis in Maude // Proc. of Workshop on Formal Methods and Security Protocols. — Indianapolis, 1998.
20. **Clavel M., Duran F., Eker S., Lincoln P., Marti-Oliet N., Meseguer J., Quesada J.** Maude: Specification and Programming in Rewriting Logic. — Menlo Park, 1999. — (Tech. Rep. / SRI International, Comput. Sci. Laboratory).
21. **Goguen J., Winkler T., Meseguer J., Futatsugi K., Jouannaud J.-P.** Introducing OBJ. — Menlo Park, 1992. — (Tech. Rep. / SRI International, Comput. Sci. Laboratory; No SRI-CSL-92-03).
22. **Zamulin A. V.** Object-Oriented Specification by Typed Gurevich Machines // Joint NCC & IIS Bull. Ser.: Comput. Sci. — 1998. — Issue 8. — P. 101—127.
23. **Zamulin A. V.** Typed Gurevich Machines Revisited // Joint NCC & IIS Bull. Ser.: Comput. Sci. — 1997. — Issue 7. — P. 93—121.
24. **Zamulin A. V.** Generic Facilities in Object-Oriented ASMs // Lect. Notes Comput. Sci. — 2000. — Vol. 1912. — P. 91—111.

Сеношенко К. О.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ СПЕЦИФИКАЦИИ: ТЕОРЕТИКО-МНОЖЕСТВЕННЫЙ И АЛГЕБРАИЧЕСКИЙ ПОДХОДЫ. ОБЗОР

Препринт
91

Рукопись поступила в редакцию 29.11.01
Рецензент И. С. Ануреев
Редактор А. А. Шелухина

Подписано в печать 10.04.02
Формат бумаги 60 × 84 1/16 Объем 2.3 уч.-изд.л., 2.5 п.л.
Тираж 50 экз.

НФ ООО ИПО “Эмари” РИЦ, 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6