COMPOSITION-NOMINATIVE SPECIFICATION LANGUAGES OF THE OBJECT-ORIENTED PROGRAMS

Abstract. The given work considers the issues of building of axiomatic specification languages of the object-oriented programs. Based on the composition-nominative method of refinement of the concept of program, axiomatic system of software specifications over the nominative data, sequential calculation of the composition-nominative logics and language Object-Z the axiomatic system of program specifications over nominative data (NDSL++) is proposed. System NDSL++ allows proving certain properties of the programs. Thus it is demonstrated that the composition-nominative approach can be effectively used for building axiomatic system of program specifications (including object-oriented) over nominative data, which quite adequately meets the problems of programming.

Keywords: formal method, object-oriented programming (OOP), software, verification, system specification

Formulas: 7; fig.: 1; tabl.: 1; bibl.: 15

JEL Classification: C6, C8

Introduction. Scope of software application includes various problem domains, including safety critical. The complexity of software systems and increasing dependence of people on their proper functioning causes increase in requirements as to reliability of the software.

Verification is the process of life cycle of ISO/IEC 12207 standard, which is responsible for verifying the correct operation of the system in accordance with its specifications and set customer's requirements.

In fact, the only verification way, traditionally used for validation of programs is testing. However, exhaustive testing of complex software systems is impossible; only small amount among all possible states of the system can be checked. As a result, testing can only find errors, not prove their absence. Thus, no program is protected against the errors, including the safety critical software.

One of the steps to solve the problem of fast and efficient design of reliable software is the use of formal methods for software development. When using formal methods it becomes possible to prove certain features of the programs, including features of correctness, by using mathematical methods.

Problem statement and literature review. Formal design methods are based on the use of language of the specifications and formal reasoning tools for building software systems. In such case the formal reasoning tools are based on languages of the formal specifications, mathematical basis and are used for analysis of the model and system. In their turn, the languages of the formal specifications are based on mathematical apparatus (predicate calculus, algebra, theory of finite automata) and used for building of the model.
Steps in the formal design are accompanied by an analysis based on the verification of system features. Check of the system features is carried out by means of formal reasoning using two basic approaches:

- automatic theorem proving - software proving of mathematical logic theorems (System Features);
- model checking - the process of check whether there is a built structure by model of given problem domain. Methods of specifications are traditionally classified by approach to representation of the model. In particular, there are:

- model-based approaches to the specification. The purpose of these specifications is to build abstract model of specified information system. These methods of specifications are based on description of the states (state-based). Examples: VDM [Jones 1990], Z [Spivey 1988, Spivey 1998], RAISE (RSL) [Björner, Henson 2008], B [Björner, Henson 2008];

A good specification must have some of the following attributes: adequate, internally consistent, unambiguous, complete, satisfied, minimal (Figure 1).

Today the absolute leader in application programming is object-oriented programming (OOP) [Booch 2004]. Thus, in the course of construction of modern languages of the program specifications it is necessary to take into account the specificity of object-oriented programming languages, in particular the fact that according to the methodology of object-oriented programming, the programs are
presented as a set of objects, each of which is an instance of a particular class, and classes form a hierarchy of inheritance [Booch 2004].

The technological base of developing various types of software is determined by the methods of object-oriented design (in particular, UML (Unified Modeling Language) [Booch, Rumbaugh, Jacobsen 1999, Ambler, Scott 2008, Warmer, Kleepe 1999]). Mentioned methods support the development of software system at all stages of the life cycle. There is a formal extension of the UML language – OCL language, which is the tool for describing additional conditions and restrictions imposed on elements of UML class diagram. OCL can be used to determine the invariants in classes, description of pre- and post-conditions in operations and methods, setting limits on the operations and as the navigation language.

Another famous model-based language of specifications, which is widespread enough for specification of the object-oriented programs developed in Java language is a behavioral language of interface specifications JML [Rodríguez, Dwyer, Flanagan, Hatcliff, Leavens, Robby 2005] based on the Larch approach [Guttag, Horning 1993].

However, like UML/OCL, JML describes the way of using the individual modules and is not intended to describe the behavior of the entire software system as a whole.

Among the formal specification languages that are able to specify object-oriented programs, independent of the development environment and can be used to describe the behavior of the entire system as a whole, one should point out Object-Z [Smith 2000], B [Björner, Henson 2008] and RSL [Björner, Henson 2008]. These languages of program specifications are based on a traditional set-theoretical approach to formalizing programs and use axiomatics of Zermelo-Fraenkel sets. The use of such developed formalism in relation to the problem of software development allows one to solve effectively specific application problems. This theory is powerful enough, but at the same time the problem of adequacy of programming (adequate semantics, data structure, and composition) is insufficient. These issues actualize the problem of search of approaches that could lead to the construction of more adequate formalisms of program specifications. We consider one of such approaches to building of axiomatic systems of non-determined program specifications [Nikitchenko, Omelchuk, Shkilniak 2006], which is based on the composition-nominative method of refinement of the concept of program [Nikitchenko 1998].

The purpose of this article study of the composition-nominative axiomatic systems for specification languages of the object-oriented programs. Achievement of this purpose is connected with the solution of the following tasks:

• analysis of methods of formal specifications;
• study of composition-nominative approach;
• development of a prototype of axiomatic system of object-oriented program specifications under the nominative data.

Object of study - the formal specification of programs, subject of study - axiomatic systems for specifications of the object-oriented programs under nominative data.

Research methods – composition-nominative method, algebraic approach, and semantic-syntactic method of filing formal models of programs are used in work.
Research results. One of the approaches to software specification is composite (exlicative) programming, initiated by academician of the NAS of Ukraine, Doctor of Physical and Mathematical Sciences, Professor V.N. Redko [Redko 1979]. Compositional programming is the development of structured logical approach by G. Frege, which was transferred to the programming by A.A. Lyapunov and developed by Yu.I. Yanov, A.P. Yershov, L.O. Kaluzhnin, V.M. Glushkov and others. Composite programming studies the systems at different levels of abstraction - abstract, Boolean and nominative (attribute) levels. Systems of the last level based on the composition-nominative methods and proposed by the Doctor of Science, Professor M.S. Nikitchenko [Nikitchenko 1998], are quite abundant for quite adequate setting of the models of data structures and programs.

Thus, the composition-nominative approach provides a single methodological basis to formalize the concept of program specification, bringing their properties and their further specification to programming languages of lower level. This approach is based on the following three principles: development, compositionality and nominativity [Nikitchenko 1998].

Development Principle (from the abstract to concrete) suggests the gradual clarification of the concept of the program, starting with the most abstract and continuing with more specific and abundant refinements.

Compositionality Principle interprets the programs as functions that are built with more features by means of special operations, called compositions.

Nominativity Principle suggests the need to use the naming relationship for programs and data operation.

Nominative data. Class of ND nominative data is constructed in the form of recursive definition: \[ ND = W \cup \left( V \mapsto ND \right) \] based on some sets of names of values and of W. Main functions over the nominative data are the following functions: naming of \( v \mapsto D_v \) and denaming of \( v \mapsto D \) with a parameter of \( v \in V \), and binary operations and predicates, such as: union \( \cup_D \), complement \( \setminus_D \), equality \( (=_W)_D \) by W. The function of construction of the empty nominative data of \( \phi_D \) predicate of membership W: \( \in \in D \) is determined. Specified operation of renaming for the nominative data \( r_D((a_1 \mapsto b_1, \ldots, v \mapsto b_v, \ldots)) = (a_1 \mapsto b_1, \ldots, x \mapsto b_x, \ldots) \). The main compositions of the functions over nominative data are binary compositions: multiplying \( \circ_D \), iteration \( *_D \), compound \( \Theta_D \) and branching ternary composition \( \diamond_D \). It is shown that the composition of multiplication corresponds to the consistent application of functions, composition of branching – to the conditional operator if-then-else of programming languages, composition of iteration \( *_D \) – to operator until-do, and composition of compound \( \Theta_D \) connects nominative data resulting from function-arguments.

The special kind of computability - nominative computability is introduced for consideration and studied in [Nikitchenko, Omelchuk, Shkilniak 2006]. Nominatively calculated functions are the functions over the nominative data obtained by locking functions \( \{ \Rightarrow_0, \Rightarrow_1, \left[ \right]_D, \setminus_D, \cup_D, (=_W)_D, \ast_D, \cap_D, \varepsilon \in D \} \) as for the multiplication of the compositions \( \{ \circ_D, \ast_D, *_D, \Theta_D \} \).

It is demonstrated in [Nikitchenko, Omelchuk, Shkilniak 2006] that an arbitrary partially recursive function can be represented by nominative computable functions over the set of natural numbers in their modeling in the class of nominative data. In addition, it is shown in [Nikitchenko, Omelchuk, Shkilniak 2006] that each nominatively calculated function can be represented by some binary \( \Sigma \) predicate \( P(x,y) \), i.e. \( f(x) = y \) if and only if \( P(x,y) \). For this
purpose the presentation of all functions specified in the definition of nominative computability are built, as well as all the functions obtained by using the compositions.

**Axiomatic theory of nominative data.** Axiomatic theory of nominative data [Omelchuk 2007] is developed in the spirit of the theory of admissible sets (S. Kripke, R. Platek, J. Barwise, Yu.L. Ershov). This theory has a number of advantages with respect to the adequacy of the programming: on the one hand, it is enough powerful to generate computable functions over the different data structures, on the other hand, it is not so restrictive as different versions of constructive logic, but it is not excessively powerful and does not allow, for example, the use of axiom of constructing the set of all subsets (compared with Zermelo-Frankel set theory). Moreover, this theory uses the basic data (urelements) corresponding to the methods of constructing data in programming. The unary predicate $U$ is used, true on the elements of the basic set $W$; the structure $(A, E_n, =, U)$ is considered. The theory of nominative data is constructed as axiomatic theory of the 1st order with equality and ternary relationship (predicate) of the nominative origin that are written in the infix form $x \rightarrow y \in_n a$ ($\in_0 (x, y) \in_n a$).

The class of $A_0$-formulas is the smallest class $Y$, containing the basic formulas and relatively closed:

1) if $\phi \in Y$, then also $\neg \phi \in Y$,
2) if $\phi, \psi \in Y$, then $\phi \land \psi \in Y$ and $\phi \lor \psi \in Y$,
3) if $\phi \in Y$, then $\forall x \rightarrow y \in_n a \in Y$ for all variables $x, y, a$.

Class of the $\Sigma$-formulas is the smallest class $Z$, containing $A_0$-formulas and closed in relation to the conditions 2) and 3) determining the class of $A_0$-formulas and further conditions of existential quantification: if $\phi \in Z$, then $\exists u \phi \in Z$.

Special axioms of the axiomatic system of program specifications over nominative data are divided into three groups:

- the first group describes the properties of equality;
- the second group of axioms describes the properties of the sets of names and data;
- the third group of axioms describes the properties of the nominative data:

  **extensionality:**
  
  $$\forall x \forall y (x \rightarrow y \in_n a \leftrightarrow x \rightarrow y \in_n b) \rightarrow a = b;$$  
  (1)

  **founding (induction by affiliation):**
  
  $$\left( \forall a \left( \forall x \rightarrow y \in_n a \phi(x) \land \phi(y) \rightarrow \phi(a) \right) \right) \rightarrow \forall a \phi(a);$$  
  (2)

  **induction by inclusion:**
  
  $$\left( \forall a \left( \forall b \subseteq a \phi(b) \rightarrow \phi(a) \right) \right) \rightarrow \forall a \phi(a);$$  
  (3)

  **$\Delta_0$-allocation:**
  
  $$\exists b \forall x \forall y \left( x \rightarrow y \in_n b \leftrightarrow x \rightarrow y \in_n a \land \phi_0 (a) \right);$$  
  (4)

  **naming:**
  
  $$\exists c x \rightarrow y \in_n c;$$  
  (5)

  **association:**


non-triviality:
\[ \exists a \exists y (x \mapsto y \in a) \]  

**Axiomatic system of program specifications over the nominative data.** Specification of the program should include a description of the program objectives, its functional structure, incoming and outgoing program data. When building the languages of program specifications it is an important task to raise the level of adequacy of the presentation of data structures, functions and compositions used in programming. In addition it is important to take into account the specifics of object-oriented programming.

In terms of composition-nominative approach, while using nominative data we can increase the adequacy of setting data structures, functions and compositions used in programming languages, and build the systems of program specifications based on the single conceptual framework. Basic data types of programming languages were specified in [Nikitchenko, Omelchuk, Shkilniak 2006], in addition, in [Nikitchenko, Omelchuk, Shkilniak 2006] the functions over nominative data were set.

Using nominative data based on composition-nominative approach we can formalize the concept of the class. Class can be represented by the nominative data as follows:

```
<class> ::= [] | <class_description>
<class_description> ::= [class \mapsto [name \mapsto <class_name>, base \mapsto [] | <class>,
interface_list \mapsto <interfaces>,
members \mapsto [attributes \mapsto <attributes>,
method \mapsto <methods>,
properties \mapsto <properties>]

<interfaces> ::= [] | <interface>, ..., <interface>
<attributes> ::= [] | <attribute>, ..., <attribute>
<methods> ::= [] | <method>, ..., <method>
<properties> ::= [] | <property>, ..., <property>

@interface> ::= [<interface_name> \mapsto <interface_realisation>]
@interface_realisation> ::= [method \mapsto <methods>, properties \mapsto <properties>]

<attribute> ::= [visibility \mapsto <visibility>, name \mapsto <nominative_data>]
<method> ::= [visibility \mapsto <visibility>, modif \mapsto <modification>;
name \mapsto <nominative_function>]
<property> ::= [visibility \mapsto <visibility>,
get \mapsto [visibility \mapsto <visibility>,
prop \mapsto <nominative_function>],
set \mapsto [visibility \mapsto <visibility>, prop \mapsto <nominative_function>]]

<visibility> ::= 0 | 1 | 2 | 3 //public, protected, private, internal
<modification> ::= 0 | 1 // virtual, override
<nominative_function> - \Delta_0^-predicate
```

Such data consist of the compositions of the types of data defined in [Nikitchenko 1998], so it is clear that the presented above objects can be set in
classes of nominative data.

The proposed presentation of the objects supports basic properties of the object-oriented programming such as inheritance, encapsulation and can maintain the polymorphism.

Given the possibility of representation of the object using nominative data we can expand suggested in [Nikitchenko, Omelchuk, Shkilniak 2006] prototype of axiomatic system of program specifications over the nominative data (NDSL), which is built on the basis of composite-nominative method of refinement of the concept of the program [Omelchuk 2007], axiomatic system of program specifications over the nominative data [Omelchuk 2007], sequential calculation over nominative data, and uses the syntax notation of language of specifications Z as a basis [Spivey 1988, Spivey 1998]. To build NDSL++ as NDSL language extension object let’s take the proposed presentation of objects as a basis, and expand NDSL language [Omelchuk 2007] by the concept of the class, similar to syntax notation of the Object-Z language [Smith 2000].

So built NDSL++ specification consists of formal mathematical text and intuitive non-formal explanation (comments). Formal text consists of sequences of sections representing scheme-classes, diagrams, global variables, basic types of the specifications. Each section is based on primes and may determine one or more scheme-classes, diagrams, basic types, global variables and global constants. It can use the names defined in other sections.

Several types of sections exist. The basic definition of the type, location status (mandatory and single), initialization scheme (mandatory and single), determination of scheme, operations, predicates, etc.

Definition of the basic types represents one or several basic types. The names used must not have previous global declaration. The area of their action extends from determination to the end of specification. Their names become part of a global dictionary of basic types.

Determination of the scheme includes type of scheme (scheme-class, state scheme, initialization scheme, and operation scheme), name, declarative and axiomatic parts. Thus, declarative part consists of a set of declarations of variables with their types. These types are the global types, or built using type constructors (Cartesian product, set, list, nominative set, and class). Axiomatic part consists of a set of $\Delta_0$-predicates [Nikitchenko, Omelchuk, Shkilniak 2006]. Definition of the scheme-class includes name, part of derivation, declarative and predicate parts, scheme of the object initialization (constructor), destructor-scheme, scheme-methods. Part of derivation may contain one section with the object of corresponding basic class and (or) several realizations of interfaces. Declarative part may contain public, protected, private and internal sections that contain respective sets of declarations of attributes with types that are global types or constructed using constructors of the types (cartesian product, set, list, nominative set, and class). Predicate (axiomatic) part consists of a set of $\Delta_0$-predicates [Nikitchenko, Omelchuk, Shkilniak 2006]. Scheme of the initialization and destructor-scheme are correspondingly the constructors and destructors of the object; scheme-method and scheme-properties are common schemes of the type of operation.

List of the predicates can appear also as a separate section. In this case, it defines properties of the specification, the implementation of which should be verified. Global variables are used in such case.

To simplify the writing and perception of NDSL++ - specification it is wise to use diagrams of UML class with extension language NDSL++. Thus one can
describe relationships between classes and interfaces, declarative part of each class in section «Attributes», and predicative part in section «Operations», while corresponding sets of $\Delta_4$-predicates for axiomatic part can be placed in sections (Body conditions, Preconditions and Postconditions). Thus, to generate NDSL++-specifications one can use standard editors of UML-diagrams (Microsoft Visio, Rational Software Architect, Software Ideas Modeler, ...). These editors retain the diagrams as XML-documents.

Thus, the system NDSL++ provides for:

- processing of incoming XML-document, representing NDSL++-specification;
- transfer of incoming text into internal representation;
- syntax checking;
- type checking;
- validation of consistency;
- check of features.

Verification of basic objects comes down to verification of the structure wherein attributes of the objects are structure data, and internal operations of the object are functions over these data.

Table 1 shows how NDSL++ compares to a few of the specification languages.

Table 1 – NDSL++ compared with other specification languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Approach</th>
<th>Completeness</th>
<th>Mathematical methods</th>
<th>Support for OOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larch</td>
<td>Algebraic</td>
<td>No</td>
<td>first-order logic</td>
<td>No</td>
</tr>
<tr>
<td>StateChar</td>
<td>Algebraic</td>
<td>No</td>
<td>automata theory</td>
<td>No</td>
</tr>
<tr>
<td>CSP</td>
<td>Algebraic</td>
<td>No</td>
<td>temporal logic</td>
<td>No</td>
</tr>
<tr>
<td>VDM</td>
<td>model-based</td>
<td>Yes</td>
<td>first-order logic</td>
<td>Yes</td>
</tr>
<tr>
<td>Z</td>
<td>model-based</td>
<td>Yes</td>
<td>first-order logic</td>
<td>No</td>
</tr>
<tr>
<td>B</td>
<td>model-based</td>
<td>Yes</td>
<td>first-order logic</td>
<td>Yes</td>
</tr>
<tr>
<td>RSL</td>
<td>model-based</td>
<td>Yes</td>
<td>first-order logic</td>
<td>Yes</td>
</tr>
<tr>
<td>UML/OCL</td>
<td>algebraic</td>
<td>No</td>
<td>automata theory</td>
<td>Yes</td>
</tr>
<tr>
<td>Object-Z</td>
<td>model-based</td>
<td>Yes</td>
<td>first-order logic</td>
<td>Yes</td>
</tr>
<tr>
<td>JML</td>
<td>algebraic</td>
<td>No</td>
<td>first-order logic</td>
<td>Yes</td>
</tr>
<tr>
<td>NSDL</td>
<td>model-based</td>
<td>Yes</td>
<td>composition-nominative logics</td>
<td>No</td>
</tr>
<tr>
<td>NSDL++</td>
<td>model-based</td>
<td>Yes</td>
<td>composition-nominative logics</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Source: compiled by the author based on own research
**Conclusions.** Based on the composition-nominative method of refinement of the concept of program, axiomatic system of software specifications over the nominative data, sequential calculation of the composition-nominative logics and language Object-Z the axiomatic system of program specifications over nominative data (NDSL++) is proposed.

System NDSL++ allows proving certain properties of the programs. Thus it is demonstrated that the composition-nominative approach can be effectively used for building axiomatic system of program specifications (including object-oriented) over nominative data, which quite adequately meets the problems of programming.

**References**


