Semantic theory of programs in next-generation intelligent computer systems

Nikita Zotov
Belarusian State University of Informatics and Radioelectronics
Minsk, Belarus
Email: nikita.zotov.belarus@gmail.com

***Abstract*—Despite the active development and usage of programming languages, currently, there is no general theory of programs on the basis of which it would be possible to design and develop applied systems. In this article, the unified ontology of programming languages and representation of programs in different programming languages is proposed. The work demonstrates the features of the representation of programs and key points of the process of their interpretation.**

***Keywords*—knowledge representation language, programming language, method representation language, software computer system, ontological approach, denotational and operational semantics of a language, procedural programming language, non-procedural programming language, ostis-systems language**

I. Introduction

For a long period of development of computer systems (c.s.), hardware restrictions on solving various problems have been practically removed. The remaining restrictions are assigned to the share of the software. First of all, these limitations are related to the current problems of software development:

- hardware complexity outstrips mankind's ability to build software c.s. using the potential capabilities of hardware;
- skills and technologies of software development lag behind the requirements for developing programs of next-generation software development;
- the ability to use existing programs is threatened by the poor quality of their development.

The key to solving these problems is a deep understanding and competent usage of existing programming languages as the main tool for the mass creation of next-generation software c.s.

This article focuses on achieving the following results:

- (1) set out the classical foundations, reflecting the accumulated world experience in the field of programming languages;
- (2) systematize the main results in this area and represent them in the form of a unified semantic theory of programs.

In this article, the problems of the current state in the field of programs and programming languages that can and should be used to develop next-generation intelligent c.s. are described in detail. It is dedicated to the basic concepts of the theory of programming languages, gives an overview of the areas for applying programming languages that are quite in demand by modern human society, describes in detail the forms and contents of criteria for evaluating the effectiveness of languages, considers ways of representing and interpreting programs of various programming languages.

II. Current state issues

In the modern era of information technologies development, there are a huge number of programming languages, each of which has its own important purpose in the field of software system design. Each language demonstrates not only its specifics but also has its own advantages and disadvantages. The variety of programming languages [1], [2] and solutions created on them is so great that it is very easy to get lost in a sea of information about all aspects of the application and design of programming languages. In addition, the main problem is not the number of existing solutions in the field of programming languages but the number of forms (!) in which specific programming languages are represented. So, declarative knowledge, i.e. knowledge that is, for example, a specification of some program, and procedural knowledge, i.e. knowledge that is programs belonging to some programming language, are represented in completely different ways, methods, and means.

In connection with the above, the following key problems in the field of programming languages can be distinguished:

1) Since the number of programming languages grows with the increase in the need for them [3], the need for describing these programming languages for further usage and design of applied systems also grows. This, in turn, requires a high level of quality in the specification of a particular language: both a description of the syntax and semantics of the constructions of this language, as well as a description of the means and methods for renovating tools that provide interpretation or translation of this

language. That is, with an increase in the number of programming languages, not only the variety of forms of knowledge representation (programming languages) grows but also the number of software systems based on various forms of knowledge representation [4].

2) A wide variety of forms of knowledge representation, as mentioned above, provides a wide range of possibilities for designing software c.s. on each of them. It turns out that in order to integrate several software systems implemented in different programming languages, it is necessary to make sure that the systems can communicate with each other in each of the languages in which they are implemented [5], [6]. Thus, the striving to use existing software components is hampered by the implementation of the components themselves [7], since in order to combine these components it is necessary to change their program code [8], [9]. The presence of a variety of forms makes it difficult to implement compatible interoperable c.s. [10].
3) As the complexity of the program code grows, the number of humans able to understand its meaning decreases. Modern developers create software c.s. without taking into account its full life cycle [11]. Systems must be constantly updated and improved with the development of the technologies on which it is based [12]. This should be ensured by good documentation of implementing the components of these systems – this reduces not only the need to raise new resources and personnel but also helps to reduce the reengineering of software c.s. [13], [9].
4) Full automation of designing software c.s. is impossible, since the modern languages in which they are designed do not have the property of reflexivity – systems cannot cognize and understand themselves [14], [15], [16] and develop almost completely on their own. Thus, the existing intelligent c.s. are not intelligent as such, since they do not have the properties they require [17].
5) The key to easy and deep mastering of a specific language as the main professional tool of a programmer is understanding the general principles of building and using programming languages [18], [19], described by their general theory. Until today, a general theory of programming languages still does not exist, which makes it difficult to develop, verify, and use new and existing programming languages. Without a general theory of programming languages, everyone can develop fundamentally general methods and tools in the way they want but not the way is required [10] - it is necessary to agree on terms and concepts and multiply the results by creating next-generation interoperable computer systems [20].
6) Achieving the maximum of services and means at a minimum of costs is possible only through a deep understanding of the principles of building programming languages due to the simplicity of means and methods of knowledge representation. The complex should be reduced to the simple and explained in simple terms, without creating an additional illusion of importance [8], [12], [21].

All these problems are related and are problems of the current state of development directions in the field of Artificial intelligence [19], [22].

So, to solve these problems, it is necessary to create comfortable conditions for the implementation of computer systems that are semantically compatible and interoperable with each other. In the context of programming languages, a general theory of designing programs for next-generation intelligent c.s. is required, which:

1) allows integrating existing solutions in the field of designing programs for computer systems without much effort and costs [23];
2) will combine knowledge representation forms of declarative and procedural types;
3) will have a wide range of tools not only for describing the syntax and semantics of existing programming languages but also for designing new analogues;
4) will be understandable not only to human but also to machine [4];
5) denotes the principles by which next-generation programming languages should be designed.

The design of such general theories, strictly speaking, must be approached with a high degree of importance. Designed c.s. should always be able to use the properties that they are drawn. In order for this theory to be used as a certain system of knowledge about how to design and use programming languages and programs in software c.s. and how to interpret their programs, it is necessary for this theory to be described by means and methods by which these software c.s. are designed. We are talking about the fact that the ontological approach [24], [4], [23], [25], [26] is a fundamentally important approach to the design of a general theory of programs.

To implement these ideas, it is necessary to study and integrate the experience gained in the field of programming languages. Therefore, the results of other researches in the field of designing the general theory of programming languages and programs will be considered below.

III. Existing ontologies of programming languages

For the most part, the ideas proposed in scientific papers on the study of programming languages are certainly in demand and useful for designing software c.s. Thus, the idea that programming languages and programs implemented on them should be organized into

a common taxonomy of concepts is fundamental, since it provides the highest quality environment for the design and implementation of c.s. The general theory of programs is needed not only to describe terms and concepts as some kind of specification used to design software c.s. (that is also important) but also in order to determine the quality of programming languages and programs on such issues as: "Is this language a programming language", "Is this knowledge a program", "How efficient is this program", "What is the degree of intelligence of this software system", etc. These ideas are proposed and discussed in the works of Raymond Turner [18], [27].

Until today, there are a large number of analogues for ontologies of programming languages and programs. The examples can be found in [28], [29]. It is also worth noting the developed ontologies of programs [14], [18], [30], [31], [32], [33], in which, strictly and unambiguously, the system of concepts is defined in formal languages – languages of logic and languages for describing the grammars of formal languages. However, none of them is such a result that could be used in the design of software c.s. without significant problems. The developed ontologies concentrate only a brief description of interconnected concepts, but the general picture of how these ontologies can be used in specific problems is almost unseen.

Today, there are completely opposite judgments about the purpose of programs and programming languages [34], which contradict the formal foundations of Artificial Intelligence [35]. There are more and more works related to the rethinking of information processing [36]. Software c.s. should not only be understandable to a human but should understand themselves, their capabilities, intentions, actions, and purposes, and understand cybernetic systems that are similar to them. Only in this way humanity and the results of its activities in the form of some specific systems will be able to work together, complementing each other and multiplying their results [10].

Based on the represented works, it can be concluded that:

- the general theory of programs and programming languages, which could be involved in solving any applied problem, as well as representing and implementing computer system design tools, has not been developed yet;
- unification of the representation of the means for description and implementation according to these descriptions as the main argument for operating the semantic knowledge representation, for complete mutual understanding between computer systems is not considered at all;
- programs and combinations of these programs in the form of program c.s. are implemented in most cases on an individual basis and are poorly documented, which complicates their usage, integration with other programs and software c.s., testing, and improvement.

The key to solving all these problems is the general technology for designing next-generation computer systems, on the basis of which it is possible to build a general theory of programs (programming discipline) [37], which will be considered further.

IV. Suggested solution

Despite the vast variety of classical technologies used by mankind, there is no general solution that allows solving the problem in a complex. Therefore, at the moment, the described problems can be solved only with the help of a general and universal solution – the OSTIS Technology. The OSTIS Technology is based on a unified version of information encoding based on semantic networks with a basic set-theoretic interpretation, called an SC-code. The language of semantic knowledge representation is based on two formalisms of discrete mathematics: set theory – defines the semantics of the language – and graph theory – defines the syntax of the language [38], [39]. Any types and models of knowledge can be described using the SC-code [40].

For the convenience of knowledge representation, there are three external knowledge representation languages based on the SC-code: SCg-code, with the help of which knowledge is displayed in the form of graph structures understandable to the average user, SCs-code, in which knowledge is represented in the form of linear text, SCn-code for displaying sc-constructions as hypertext. This representation is close to natural, understandable to the average user [40].

The OSTIS Technology is suitable for solving the listed problem, since:

1) The Standard of the OSTIS Technology [40] already implements the basic tools necessary for the design and development of interoperable c.s., which are based on the semantic knowledge representation. This eliminates not only the need to create top-level ontologies, which should be used in the general theory of programs as the basis for describing the concepts of this theory, but also helps to design solutions consistent with other ontologies. As a result, a common coherent world picture is formed, which is (1) consistent, that is, agreed, (2) unambiguously interpreted, (3) universal, and, (4) most importantly, understandable to everyone.
2) The OSTIS Technology is designed by a single unified knowledge representation language called an SC-code. The meaning of programs and programming languages is understandable and unambiguous if and only if this meaning is described in one common language understandable to any cybernetic system. The meaning lies not in the syntax of the signs, but

in the configuration of the connections between them (!) [40], [41], [42].
3) The SC-code is syntactically minimal. The minimum number of signs is used to describe objects and connections between them. At the same time, the diversity of these connections is reduced to the diversity of sign constructions. All this is provided by representing information in the form of graph structures [43], [44], [45].
4) The SC-code is not just convenient for describing and designing some complex objects – it can be used to design and implement any knowledge representation languages, including programs, computer systems, and, in general, the real world.
5) Ontological [46], [26] and component [47] approaches to the design of any complex objects ensure the fulfillment of the main principles by which modern systems should be designed. What is implemented and can be used, must be reused everywhere [48], [49].

Thus, the solution to all described problems is the general theory of programs, interpreted as an ontology of the general system, implemented through the OSTIS Technology.

V. General description of designed subject domains and ontologies

The result of this work is a *Subject domain and ontology of methods* (Subject domain and ontology of programs), which can be used to set methods (programs), their syntax, denotational and operational semantics. The *Subject domain and ontology of methods* is a private subject domain in relation to the *Subject domain and ontology of information constructions and languages*. This means that it inherits all the properties of the concepts and relations studied in it.

Subject domain and ontology of information constructions and languages
⇒ *private subject domain**:
- *Subject domain and ontology of languages*
  ⇒ *private subject domain**:
  - *Subject domain and ontology of natural languages*
  - *Subject domain and ontology of formal languages*

Subject domain and ontology of formal languages
⇒ *private subject domain**:
- *Subject domain and ontology of knowledge representation languages*
  ⇒ *private subject domain**:
  - ***Subject domain and ontology of methods***

Subject domain and ontology of methods
⇒ *private subject domain**:
- *Subject domain and ontology of methods of ostis-systems*
  ⇒ *private subject domain**:
  - *Subject domain and ontology of procedural methods of ostis-systems*

∋ *maximum studied object class'*:
- *method*

∋ *non-maximum studied object class'*:
- *method representation language*
- *method class*
- *meta-method*
- *process*
- *variable*
- *constant*
- *operator*
- *method quality*

∋ *explored relation'*:
- *submethod**
- *subprocess**
- *method syntax**
- *parameter'*
- *start operator'*
- *denotational semantics of the method**
- *operational semantics of the method**
- *method of the specified method representation language**

VI. Concept of a method (program)

Each theory must be conceptually consistent. Despite the fact that there are different interpretations for the concept of a programming language in the literature, there should be a universal one. To do this, instead of programming languages, we will further talk about method representation languages and instead of programs of these programming languages – about methods as sign constructions of method representation languages (m.r.l.). This decision is justified by the fact that usually the language acts as a tool for some kind of knowledge of a certain type, and the term of the programming language is degenerate, since it is worth talking not about languages in which something can be programmed but about languages in which knowledge of a certain type can be represented, in this case – knowledge of a procedural kind. The terms of the programming language and the program themselves will be considered as non-basic identifiers for the concepts of the methods and method representation language, respectively.

Formally, a *method* is a specification for solving a problem of some class [40], [50]. The specification of each class of problems includes a description of the "binding" of the method to the initial data of a particular problem solved with the help of this method.

method
:= [program]
:= [description of how any or almost any action belonging to the corresponding action class can be performed]
:= [method for solving the corresponding class of problems that provides a solution to any or most problems of the specified class]
:= [generalized specification for solving problems of the corresponding class]
:= [program for solving problems of the corresponding class, which can be either procedural or declarative (non-procedural)]
:= [knowledge of how to solve problems of the corresponding class]
⊂ *knowledge*
∈ *knowledge type*
:= [way]
⊃ *problem-solving model*

VII. Concept of a method class. General classification of methods

Sometimes, it may be appropriate to allocate a certain subset of methods (for example, a set of methods with which a certain problem is solved), then in this case for these methods it is possible to describe the requirements that they must fulfill. Such sets of methods are *method classes* of some m.r.l., which are associated with a particular *problem-solving model*. Methods can be either *procedural* or *non-procedural* [18].

method class
⇐ *family of subclasses**:
    method
:= [set of methods for which the representation (specification) of these methods can be unified]
:= [set of various problem-solving methods that have a common language for representing these methods]
:= [set of methods for which the representation language of these methods is set]
∋ *procedural problem-solving method*
    ⊃ *algorithmic problem-solving method*
∋ *non-procedural problem-solving method*
    ⊃ *logical problem-solving method*
    ⊃ *production problem-solving method*
    ⊃ *functional problem-solving method*
        ⊃ *artificial neural network*
        ⊃ *genetic "algorithm"*
:= [set of methods, which is associated with a particular problem-solving model]

Since each method corresponds to a generalized formulation of the problems solved using this method, each method class must correspond not only to a certain m.r.l. belonging to the specified *method class* but also to a specific language for representing generalized formulations of problems for different classes of problems, solved by methods belonging to the specified method class.

For procedural and non-procedural methods, although it is possible to set *input* and *output parameters*, the general denotational semantics of their logical elements cannot be set: for procedural methods, these are operators, for non-procedural methods – mathematical objects of the subject domain.

VIII. Concept of method representation language (programming language)

Each specific method class corresponds one-to-one to the m.r.l. belonging to this (specified) method class. Thus, the specification of each method class is reduced to the specification of the corresponding m.r.l., that is, to the description of its syntactic, denotational, and operational semantics. Examples of m.r.l. are all programming languages that basically belong to the subclass of m.r.l., but now the need to create effective formal m.r.l. for performing actions in the external environment of cybernetic systems is becoming increasingly important. Without this, complex automation [51], in particular, in the industrial sector, is impossible.

By *method representation language* we mean a formal language, (1) the sign constructions of which are the corresponding methods for which there are general building rules and (2) general rules for correlating with those entities and relations between them that are described by these methods.

With the help of m.r.l., *messages* (methods) for the computer are generated. These messages must be understandable (semantically correct and consistent) to the computer [52].

method representation language
:= [programming language]
⊂ *knowledge representation language*
    ⊂ *formal language*
:= [computer language]
:= [formal language, (1) the symbolic constructions of which are the corresponding methods for which there are general building rules and (2) general rules for correlating with those entities and relations between them that are described by these methods]
:=

[mean of communication between a human (user) and a computer (performer)]
:= [tool for producing software services]

A method belongs to a method representation language if it is a syntactically correct, syntactically consistent, semantically correct, and semantically consistent method of the specified m.r.l. (!).

relation set in multiple method representation languages^
:= [relation whose scope of definition includes many different method representation languages]
∋ *method of the specified method representation language**
∋ *syntactically correct method for the specified method representation language**
  := [method that does not contain syntax errors for the specified method representation language*]
  ⊂ *syntactically correct sign construction for the specified language**
∋ *syntactically consistent method for the specified method representation language**
  ⊂ *syntactically consistent sign construction for the specified language**
∋ *semantically correct method for the specified method representation language**
  := [method that does not contain semantic errors for the specified method representation language*]
  ⊂ *semantically correct sign construction for the specified language**
∋ *semantically consistent method for the specified method representation language**
  ⊂ *semantically consistent sign construction for the specified language**
  := [method of the specified method representation language that contains sufficient information to determine its truth*]

method of the specified method representation language*
:= [method belonging to the specified programming language*]
⊂ *text of the specified language**
⇒ *second domain**:
method
⇐ *combination**:
{• {}
  ⇐ *combination**:
  {• *syntactically correct method for the specified method representation language**
   • *syntactically consistent method for the specified method representation language**
  }
 • {}
  ⇐ *combination**:
  {• *semantically correct method for the specified method representation language**
   • *syntactically consistent method for the specified method representation language**
  }
}

IX. General classification of method representation languages

In the modern information society, method representation languages (m.r.l.) are distinguished by their paradigms: *procedural*, *functional*, *logical*, *object-oriented* m.r.l., etc. The solution of the problem by the computer is made in the form of a sequence of operators: in the methods of functional m.r.l. – indication of other methods; in logical m.r.l., operators are used; and in object-oriented ones – objects.

method representation language
⊃ *general-purpose method representation language*
  := [general-purpose programming language]
⊃ *subject-oriented method representation language*
  := [subject-oriented programming language]
⇒ *subdividing**:
*method representation language paradigm^*
  = {• *procedural method representation language*
     • *non-procedural method representation language*
    }

Procedural method representation languages set computations as a sequence of operators (commands). They are focused on computers with von Neumann architecture. Basic concepts of procedural m.r.l. closely related to computer components:

- variables of various types that model computer memory cells;
- assignment operators that model data transfers between memory areas;
- repetitions of actions in the form of iteration, which simulate the storage of information in adjacent memory cells;
- and more.

procedural method representation language
:= [imperative method representation language]
⊃ *structural method representation language*
  ∋ *example*$'$:
    • *Fortran*
    • *C*
    • *Pascal*
⊃ *object-oriented method representation language*
  ∋ *example*$'$:
    • *Smalltalk*
    • *Java*
    • *HTML*
  ⊃ *aspect-oriented method representation language*
⊃ *script method representation language*
  := [patch method representation language]

Non-procedural method representation languages, in contrast to procedural languages, set computations as a sequence of interconnected objects. Basic concepts of non-procedural m.r.l. usually are not related to computer components.

non-procedural method representation language
:= [declarative method representation language]
⊃ *logical method representation language*
  ∋ *example*$'$:
    • *Prolog*
⊃ *production method representation language*
⊃ *functional method representation language*
  := [applicative method representation language]
  ∋ *example*$'$:
    • *LISP*

X. Representation of the syntax and semantics of various methods

The *syntax* and *semantics* of a method represent its *specification*. The semantics of a method can be viewed from two perspectives: as a set of interrelated knowledge, which is determined by the denotational semantics of this method, and as knowledge that can be interpreted by another method, which is determined by the operational semantics of this method.

method specification*
⇒ *subdividing**:
  {• *method syntax**
  • *denotational semantics of the method**
    := [generalized formulation of the class of problems solved using this method*]
    ⇔ *semantically close sign**:
      *generalized formulation of the problems of the corresponding method class**
  • *operational semantics of the method**
    := [list of generalized agents providing method interpretation*]
    := [family of methods for interpreting this method*]
    := [formal description of the specified method interpreter*]
  }

A. Representing the syntax of the problem-solving method

Any method consists of atomic information constructions that set the order of actions in the knowledge base, with the help of which it is required to move from the initial state to the target one, thus solving some specific problem. So, for example, in a procedural method, any such operator represents some mathematical function. Expressions and operators are used to compose these functions into larger fragments. In turn, linear sequences of operators and conditional branches can also be represented by functions composed of functions inherent in particular components of these constructions. A cycle is easily described by a recursive function composed of the components included in its body.

The *method syntax** defines the set of its allowed constructions. The appearance of method elements is specified using a certain syntax. It describes such lexical details as the location of keywords and punctuation marks. Grammars are used to specify a particular syntax.

The syntax of m.r.l. in ostis-systems can be formally described in various ways. So, for example, it is possible to use the Backus-Naur meta-language to describe the syntax of some methods of a particular m.r.l. Other equally well-known forms of method representation are context-free grammars, extended Backus-Naur form, syntactic graphs [1], [53], [54].

However, it is much more logical and advisable to describe the syntax of other languages in the universal knowledge representation language – the *SC-code*. This approach will allow ostis-systems to independently understand, analyze, and generate texts of these languages on the basis of principles common to any form of external information representation, including non-linear ones [45]. Thus, languages written in the SC-code have the same syntax as the SC-code.

B. Representing the denotational semantics of the method

The semantics of a method explains the meaning of the syntactic constructions of a method. The most common methods for describing the semantics of programming languages are: denotational, operational, axiomatic, algebraic ones [55], [56]. Based on the principles of the OSTIS Technology, by the semantics of a method we

mean the combination of the denotational and operational semantics of the method.

The description of how to "bind" a method to some class of problems includes:

- a set of variables that are included both in the method and in the generalized formulation of the problems of the corresponding class and whose values are the corresponding elements of the initial data of each specific problem being solved;
- part of the generalized formulation of problems of the class to which the method under consideration corresponds, which are a description of the conditions for applying this method;
- a description of the method initiation condition and its result;
- a description of initial and target situations in sc-memory.

"Binding" a method to a specific problem solved with the help of this method is carried out by searching for such a fragment in the knowledge base, that satisfies the conditions for applying the specified method. One of the results of such a search is the setting of a correspondence between the above variables of the method used and the values of these variables within a specific problem being solved. Another option for setting the correspondence under consideration is an explicit call of the corresponding method (program) with an explicit transfer of the corresponding parameters. However, this is not always possible, since when executing the process of solving a specific problem based on the declarative specification for performing this action, it is not possible to identify:

- when it is necessary to initiate a call (usage) of the required method;
- which specific method to use;
- which parameters, corresponding to the particular problem being initiated, must be passed in order to “bind” the method used to this problem.

A *process* is understood as some action in sc-memory that unambiguously describes a specific act of executing a certain method for given initial data [37]. If a method describes an algorithm for solving a problem in general terms, then a process denotes a specific action that implements this algorithm for given input parameters. In fact, the process is a unique copy created on the basis of a method in which each sc-variable corresponds to a generated sc-constant.

relation defined on a set (process)^
:= [relation whose scope of definition includes many possible processes]
∋ *parameter’*
⇒ *subdividing**:
{• *in-parameter’*
• *out-parameter’*
}
∋ *in-parameter’*
∋ *out-parameter’*
∋ *initial information construction’*
∋ *subprocess**

The process of “binding” a problem-solving method to a specific problem solved using this method can also be represented as a process consisting of the following phases:

- building a copy of the used method;
- pasting the main (key) variables of the method used together with the main parameters of a specific problem being solved.

As a result, on the basis of the considered method used as a sample (template), a specification of the process for solving a specific problem is built. The description of the process of “binding” the solution method to a specific problem, as well as the description of the elements of the method, is the *denotational semantics of this method*.

denotational semantics of the method
∋ *general formulation of the class of problems**
:= [text formulation of the set of problems solved by this method]
⊂ *explanation**
∋ *primary initiation condition**
∋ *initiation condition and result**
⇐ *Cartesian product**:
⟨• *method class*
• *implication**
⟩
∋ *condition of initial and target situations**
⇐ *Cartesian product**:
⟨• *method class*
• *implication**
⟩

An example of the part of the specification that describes the denotational semantics of the Method for finding the double sum of two numbers is demonstrated in Figure 1.

The *general formulation of the class of problems** relation is a class of sc-connectives between an sc-connective, denoting a set of methods, and an ostis-system file, which is an explanation of which classes of problems can be solved using a given set of methods. In some rare cases, the presence of such an sc-connective may not be in the specification of a method, since there is no need to specify which classes of problems can be solved using this method.

The connectives of the *primary initiation condition** relation connect the sc-connective, denoting a set of methods, and the binary oriented pair, describing the primary condition for initiating a given method, i.e. such a

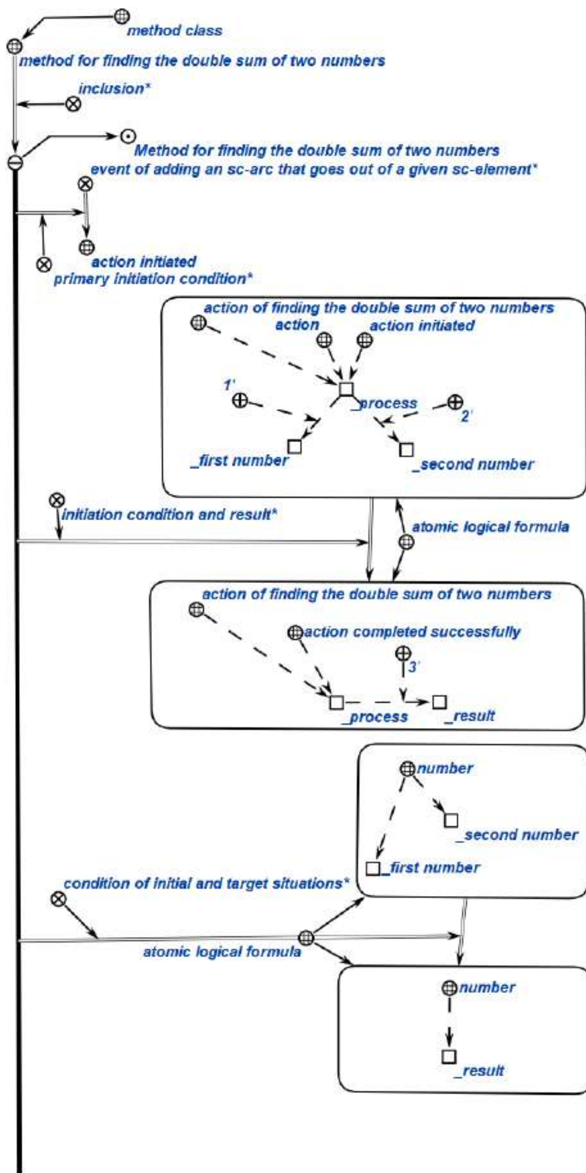

Figure 1. The specification of a method for solving the problem of calculating the double sum of two numbers

specification of the situation in sc-memory, the occurrence of which prompts the meta-method-executor to transfer the given set of methods into the active state and begin checking for their full initiation condition.

The first component of this oriented pair is the sign of some class of elementary events in sc-memory*, for example, the event of adding an sc-arc going out of a given sc-element*.

In the general case, the second component of this oriented pair is a random sc-element, with which the specified type of event in sc-memory is directly associated, i.e., for example, the sc-element, from which the generated or deleted sc-arc or file, the contents of which have been changed, goes out, or in which this sc-arc or the file come.

The connectives of the initiation condition and result* relation link together the sc-connective, denoting the set of methods, and a binary oriented pair, linking the initiation condition for this set of methods and the results of executing this set of methods in any particular system. The specified oriented pair can be considered as a logical implication connective, while the universality quantifier is implicitly imposed on sc-variables present in both parts of the connective and the existence quantifier is implicitly imposed on sc-variables present either only in the premise or only in the conclusion.

The first component of the specified oriented pair is a logical formula that describes the condition for initiating the described method, that is, the construction, the presence of which in sc-memory calls a lot of methods to start working on changing the state in sc-memory. This logical formula can be both atomic and non-atomic, which allows using any connectives of the logical language.

The second component of the specified oriented pair is a logical formula that describes the possible results of performing the described set of methods, that is, a description of the changes in the state of sc-memory made by it. This logical formula can be both atomic and non-atomic, which allows using any connectives of the logical language.

The connectives of the *condition of initial and target situation** relation connect an sc-connective, denoting a set of methods, and a binary oriented pair, connecting the initial and target situations in sc-memory, that is, in short, the situation before applying the method and the desired situation after applying the method. The specified oriented pair can also be considered as a logical implication connective, while on the sc-variables present in both parts of the connective the universal quantifier is implicitly imposed, and on the sc-variables present either only in the premise or only in the conclusion the existential quantifier is implicitly imposed. For the first and second components of the specified oriented pair, the same restrictions and properties are imposed as for the components of the oriented pair, which is the second component of the *initiation condition and result** relation.

It should be noted that the connectives of the *initiation condition and result** relation and the *condition of the initial and target situation** relation can be represented differently. Sometimes, it may not be necessary to create and check the second condition of the method, which checks for the presence of the initial situation in sc-memory and checks for reaching the target situation in sc-memory as a result of applying the method. If so, then the condition of the initial and target situation* can be specified in the logical formulas that are components in the second component of the connective of the *initiation condition and result** relation.

Programs, depending on the way of their representation

in languages, will differ. This can be verified by comparing examples of procedural (Fig. 2) and logical (Fig. 3) methods for solving the same problem.

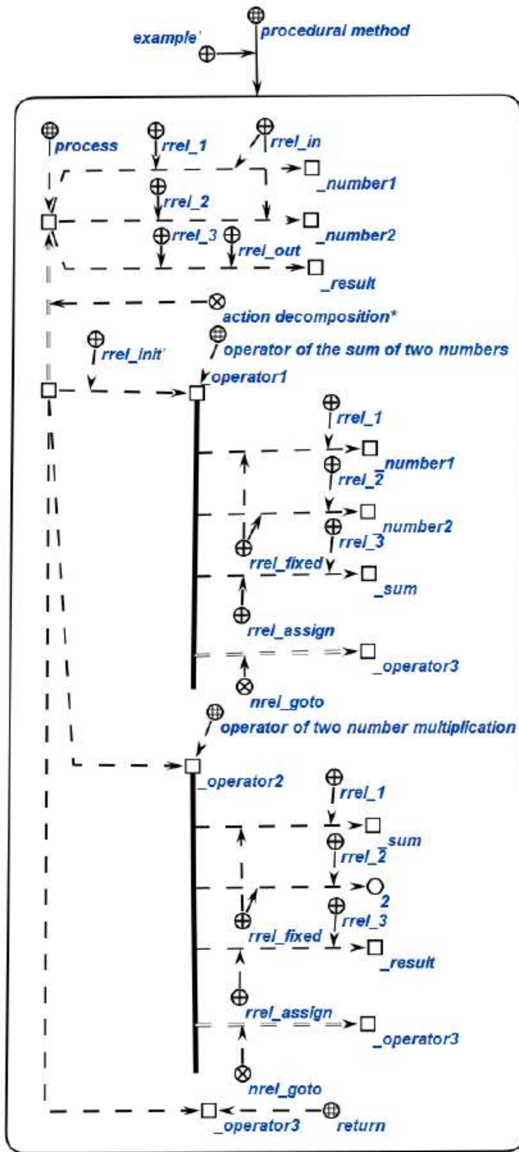

Figure 2. An example of a procedural method for solving the problem of calculating the double sum of two numbers

With the help of the SC-code, it is also possible to represent those languages that are not written in it. The problem will be in the fact that the form and meaning of the language and its methods will be separated, that is, they will be represented in different ways. In this case, the SC-code is a powerful tool for integrating the specifications of various languages of external knowledge representation. However, it should be noted that there is no need to represent different forms of methods belonging to different method representation languages within the OSTIS Technology. This is explained by the following

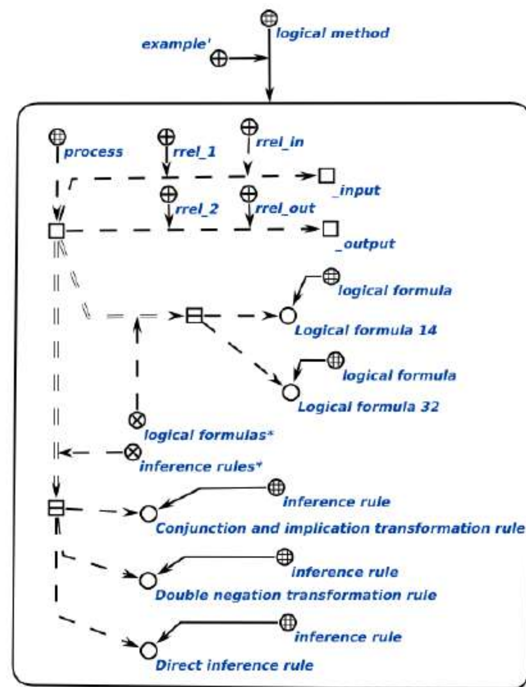

Figure 3. An example of a logical method for solving the problem of calculating the double sum of two numbers

facts:

1) The SC-code is a fairly universal language for representing any kind of knowledge. This means that different forms of the algorithm for solving the same problem can be minimized. In the SC-code, the foundation is a formal theory, which provides a universal representation of various types of declarative and procedural knowledge. Thus, logical methods can be represented as procedural programs, in which as operands of operators not only logical formulas and inference rules will serve but also other methods that provide interpretation of these logical formulas using inference rules. Thus, the SC-code can be called not only a language of unified knowledge representation but also a language in which different classes of problems can be solved in the same way.
2) Various types of knowledge in ostis-systems, designed according to the principles of the OSTIS Technology, are deeply integrated with each other. This provides not only simplicity for creating these systems based on existing languages that can be described in the SC-code but also great opportunities for creating basic programming languages for next-generation computer systems, such as, for example, the basic language for representing SCP procedural methods, the basic language for representing production methods, etc. Modern method representation languages are created to simplify the description of

some algorithm for fast and high-quality solution of a certain class of problem [57]. In turn, the proposed methods and models make it possible to design an m.r.l. for next-generation computer systems with the help of basic knowledge representation languages in such a way that the very form of knowledge representation does not change. Methods of different m.r.l. must have one universal form of representation, i.e. the same syntax, but may allow the denotational and operational semantics of their methods to be described and represented in different ways using the same syntax.

3) Designing new m.r.l. should be reduced to their full description in the minimum family of SC-code languages: the SC-code itself, SCP, and SCL. We are talking about designing a new method representation language: it is enough to develop a (non-atomic) meta-method in SCP and SCL languages, which will interpret the methods of the languages being designed and also describe the denotational semantics of these methods. Meta-method for interpreting m.r.l. methods can be called an interpreter of these languages, that is, some abstract sc-machine on which it is possible to execute methods of a certain language for representing these methods.

C. Representing the operational semantics of the method

A complete *method specification**, in addition to the *denotational semantics of this method**, must include *the operational semantics of this method**, that is, a formal description of the interpreter of the given method. Operational semantics of the m.r.l. describes the execution of a method written in a given language by means of a virtual computer. A virtual computer is defined as an abstract automaton. The internal states of this automaton model the states of the computational process when the method is executed. The automaton translates the source text of the method into a set of formally defined operations. This set defines the transitions of the automaton from the initial state to the sequence of intermediate states by changing the values of the method variables. The automaton completes its work by passing to some final state. Thus, here we are talking about a fairly direct abstraction of the possible usage of m.r.l. Operational semantics describes the meaning of a method by executing its operators on a simple automaton. The changes that occur in the state of the machine, when a given operator is executed, determine the meaning of that operator.

The operational semantics of a specific method is reduced to the description of a *meta-method* that interprets it, verifies, etc.

meta-method
⊂ *method*
:= [method whose parameter values are other methods]

operational semantics of the method
∋ *interpretation meta-method**
  ⇐ *Cartesian product**:
    ⟨• *method class*
     • *method*
    ⟩
∋ *meta-method for verification and quality assessment**
  ⇐ *Cartesian product**:
    ⟨• *method class*
     • *method*
    ⟩

The *interpretation meta-method** relation is a class of sc-connectives between an sc-connective, denoting a set of methods, and an sc-node, denoting a method that is capable of interpreting a given set of methods. The *meta-method of verification and quality assessment** is a class of sc-connectives between an sc-connective, denoting a set of methods, and an sc-node, denoting a method that is capable of verifying and evaluating the quality of a given set of methods.

Within the OSTIS Technology, there can be a wide variety of such meta-methods. Each of them can consist of many atomic and non-atomic submethods. These can be both meta-methods that interpret the methods of certain m.r.l. and meta-methods that verify and analyze the quality of these methods. In addition, meta-methods can perform operations on other meta-methods.

meta-method for methods interpreting base method representation languages
⇒ *inclusion**:
- *meta-method for methods interpreting the SCP procedural method representation language*
- *meta-method for methods interpreting the SCL logical method representation language*
- *meta-method for methods interpreting the production method representation language*
- *meta-method for methods interpreting the functional method representation language*
- *meta-method for methods interpreting the neural network representation language*
- *meta-method for methods interpreting the representation language of genetic algorithms*

meta-method for verifying and evaluating the quality of methods in basic method representation languages
⇒ *inclusion**:
- *meta-method for verifying and evaluating the quality of methods in the SCP procedural methods representation language*
- *meta-method for verifying and evaluating the quality of methods in the representation language of logical SCL methods*
- *meta-method for verifying and evaluating the quality of methods in the representation language of production methods*
- *meta-method for verifying and evaluating the quality of methods in the representation language of functional methods*
- *meta-method for verifying and evaluating the quality of neural network representation language methods*
- *meta-method for verifying and evaluating the quality of methods for the representation language of genetic algorithms*

The concepts of syntax, denotational and operational semantics of method representation languages are reduced to the concepts of syntax, denotational and operational semantics of any language in general.

XI. Representation of the syntax and semantics of method representation languages

It is clear that in order to use the m.r.l., each language construction should be described separately, as well as its usage in aggregate with other constructions. There are many different constructions in a language, the exact definition of which is necessary both for the programmer using the language and for the developer of the compiler for that language. This knowledge allows the programmer to predict the calculations performed by the method operators. The constructions descriptions are necessary for the developer to create a correct implementation of the compiler.

A description of a formal model of a method representation language can be given by its *specification*. The specification contains a description of the syntax and semantics of the m.r.l.

method representation language specification*
⊃ *relation posed on a set (method representation language)**
⇒ *subdividing**:
{• *syntax of the method representation language**
  ⊂ *language syntax**
  := [be a theory of well-formed information constructions belonging to a given method representation language]
• *denotational semantics of the method representation language**
  ⊂ *language denotational semantics**
  := [generalized formulation of the classes of problems solved using this method representation language*]
• *operational semantics of the method representation language**
  ⊂ *language operational semantics**
  := [list of generalized agents that provide interpretation of methods of a given method representation language*]
  := [family of methods for interpreting texts in a given method representation language*]
  := [formal description of the interpreter of the specified method representation language*]
}

The *syntax of m.r.l.** is a binary oriented relation, each pair of which associates a sign of some language with a description of syntactically allocated classes from fragments of constructions of a given m.r.l. with a description of relations defined on these classes and with conjunction of quantifier propositions, which are the syntactic rules of the given language, that is, the rules that all syntactically correct (well-built) constructions of the specified m.r.l. must satisfy. In the general case, the *syntax of the m.r.l.** relation is no different from the *language syntax** relation, but still there is a refinement, since m.r.l. are languages in general, and the syntax of the m.r.l. inherits all syntax properties of any languages. The *syntax of the m.r.l** combines the syntaxes of all methods belonging to a given method representation language.

*Denotational semantics of m.r.l.** means a binary oriented relation, each pair of which associates a sign of some language with the sign of some ontology, which can be used to describe the methods of this language, and *operational semantics of m.r.l** is a description of the meta-method for interpreting the methods of this language.

In the context of this work, specific types of denotational and operational semantics will not be considered further.

XII. Help-system for design and method development support

The current state of the art in software design and development suggests that developers are more eager to automate the development of methods in specific method representation languages than to provide training tools for their design, including the design of new method representation languages. This leads to the following problems:

1) While the number of developers who understand the code of a complex software system is decreasing, the requirements for that system are growing faster and faster. Often, developers of complex software systems themselves are not able to explain the logic of these systems. For this reason, it is necessary to create tools that will automate the documentation of software systems [52].
2) To train new developers in the skills of working with software systems and their development, it is necessary to attract the resources of development experts who understand the principles of operation of these software systems. The problem is solved by developing a help system that will not only teach the user how to design problem solving methods and software systems based on these methods, but also point out gaps in related disciplines necessary to achieve high-quality results of all their activities.
3) In engineering, developers often design and develop solutions that have already been created by other specialists. Thus, functionally equivalent methods of solving problems are obtained, and even software systems that solve similar problems. The key to solving this problem is to design a semantically powerful library of reusable problem solving methods.

Thus, the semantic theory of programs alone is not enough. In addition to it, for a permanent and unhindered design and development of methods of a different class, it is necessary to develop:

1) an intelligent help system for supporting the design and development of methods, mentioned in [58], which will not only help the developer verify the method being developed, but also suggest ways to develop it;
2) a semantically powerful library of reusable components [47] for quickly finding existing problem solving methods and applying them to other more complex problems [46].

The potential help system should be part of a common development tool for next-generation intelligent computer systems - ostis-platform [59] - and may consist of the following components:

- the intelligent help-system on the semantic theory of programs;
- the intelligent help system on the library of reusable problem solving methods,
- the intelligent help-system for a set of tools for designing methods for solving problems,
- the intelligent help-system on the methodology of teaching the design of various methods for solving problems.

Each component contains knowledge from the relevant area of design and development theory of problem solving methods. In accordance with open semantic technology, each component must include:

- reference subsystem,
- subsystem for monitoring and analyzing the activities of the developer of methods for solving problems,
- learning management subsystem.

Each of the subsystems interacts with other subsystems and can also function autonomously.

The reference subsystem is an expert consultant in the field of semantic program theory who can answer any question from a novice or experienced user. Each of these systems can become individual assistants in the training of new specialists - a personal ostis-assistant. The functions of the reference subsystem include:

- search for information at the request of the user, including freely-designed ones;
- displaying the information found, taking into account the user's skill level;
- analysis of program texts and making suggestions to improve their effectiveness;
- generation of program texts on request to the user;
- self-initiation in case of difficulties for the user or the student.

Thus, the development of such components according to the principles of the OSTIS Technology will confirm the general semantic theory of programs.

XIII. Quality (efficiency) criteria of methods

The method representation language can be defined by a set of indicators that characterize its individual properties. The problem arises of introducing a measure to assess the degree of suitability of the m.r.l. to the performance of the functions assigned to it – *method quality* [6], [56], [60]. The quality criteria of methods are given on the basis of particular indicators of the efficiency of these methods (quality indicators). The method of connection between particular indicators determines the type of efficiency criterion.

method quality
⇒ *prerequisite property**:
- *ease of reading and understanding the method*
- *ease of creating the method*
- *method cost*
- *total volume of problems solved using this method class*

- *variety of types of problems solved using this method class*
- *method reliability*

Ease of reading and understanding the method should make it easy to highlight the basic concepts of each part of the method without referring to its specification.

ease of reading and understanding the method
⇒ *prerequisite property**:
  - *m.r.l. syntax simplicity*
  - *orthogonality of m.r.l. information structures*
  - *structured flow of control in a method*

The method representation language should provide a *simple* set of informational constructions that can be used as basic elements when creating methods. The syntax of the language has a strong impact on simplicity: it must transparently reflect semantics of constructions, exclude ambiguity and non-uniqueness of interpretation.

Orthogonality means that any possible combination of different information constructions will be meaningful, with no unexpected behavior resulting from the interaction of the constructions or context of usage.

The order of control transfers between method operators, i.e. the *flow of control*, should be human readable and understandable.

Ease of creating the method reflects the convenience of the language for representing that method in a particular subject domain.

ease of creating the method
⇒ *prerequisite property**:
  - *m.r.l. syntax simplicity*
  - *m.r.l. natural syntax*
  - *orthogonality of m.r.l. information structures*
  - *completeness and accuracy of m.r.l. specification*
  - *consistency and integrity of m.r.l. specification*

The syntax of the method should facilitate an easy and transparent display of the algorithmic structures of the subject domain in it. The syntax of m.r.l. should be not only *simple*, but also *natural*, and support the *orthogonality* of language informational constructions.

Ease of representation of a new method is ensured by *complete and precise, consistent and integral specification* of the appropriate language. That is, it is required to have a sufficient number of information constructions in this language in order to represent a particular method. At the same time, the language specification must be consistent and integral in order to represent consistent methods.

Cost of the m.r.l. method is made up of several components.

method cost
⇒ *prerequisite property**:
  - *cost of method applying*
  - *cost of method interpretation*
  - *cost of method creating, testing, and using*
  - *cost of method maintenance*

Cost of method applying largely depends on the structure of the m.r.l. A language that requires numerous syntactic type checks during method application will prevent the program from running quickly.

Cost of method interpretation depends on the capabilities of the interpretation meta-method used. The more perfect the optimization methods are, the more expensive will be the interpretation costs. The amount of the cost of creating, testing, and using the method depends on the used meta-method of verification and evaluation of the quality of this method.

Numerous studies show that a significant part of the cost of the method used is not the cost of its development but the *cost of its maintenance* [11]. Associating method maintenance with other method characteristics, the dependence on readability, since maintenance usually occurs by the next generation of developers, should first of all be highlighted.

The total volume of problems and the variety of types of problems solved with the help of this method class are no less important characteristics, which show the degree of universality of the corresponding m.r.l. The more problems can be solved on m.r.l., the more universal it is.

Reliability of m.r.l. methods should be ensured by a minimum of errors during the operation of a particular method.

All of these criteria can be applied to the method representation languages themselves.

XIV. Directions of development

This article is the beginning of the semantic theory of programs for next-generation c.s. The logical development of this work will be:

- refinement and addition of concepts of the *Subject domain and ontology of methods* to achieve the completeness of the theory;
- description of private subject domains of the *Subject domain and ontology of methods* for specific types of methods, as well as clarification of the denotational and operational semantics of the specification of these methods;
- description of possible ways of implementing meta-methods for interpreting methods of various m.r.l;
- implementation of tools to support the design and development of various methods for solving the

problem and the development of their respective specifications;
- formalization of mathematical models for calculating method efficiency estimates.

XV. Conclusion

The main conclusion of this work is that it is necessary not to replenish knowledge about which programming languages already exist and to reveal possible areas of their application, but to develop fundamentally new programming languages with which it was possible to create next-generation intelligent computer systems with high level of intelligence, semantic compatibility and interoperability with similar computer systems, unification of knowledge representation and processing, platform independence from tools for their implementation, and so on.

Such systems should be developed according to the principles of the OSTIS Technology, and their main development languages will be graph languages for representing methods that are sublanguages in relation to the basic procedural programming language SCP.

In this article, the problems of ensuring the design of software systems are considered. A comparative analysis of existing solutions in the field of unifying the representation of programming languages has been carried out. The work defines the solution of the problem in the form of designing and developing a universal theory of programming languages according to the principles underlying the OSTIS Technology. This article is also a specification of how software systems should be specified and designed.

Acknowledgment

The author would like to thank the research groups of the Departments of Intelligent Information Technologies of the Belarusian State University of Informatics and Radioelectronics and the Brest State Technical University for their help in the work and valuable comments.

References

[1] Sebesta, R. W, *Concepts of Programming Languages*. 10th ed. — Pearson/Addison-Wesley, 2012.
[2] Tourlakis, George, *Computability*. Springer Nature, 2022.
[3] A. Iliadis, "The tower of babel problem: making data make sense with basic formal ontology," *Online Information Review*, vol. 43, no. 6, pp. 1021–1045, 2019.
[4] C. M. Zapata Jaramillo, G. L. Giraldo, and G. A. Urrego Giraldo, "Ontologies in software engineering: approaching two great knowledge areas," *Revista Ingenierías Universidad de Medellín*, vol. 9, no. 16, pp. 91–99, 2010.
[5] Golenkov, V., Guliakina, N., Davydenko, I., Eremeev, A., "Methods and tools for ensuring compatibility of computer systems," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, V. Golenkov, Ed. BSUIR, Minsk, 2019, pp. 25–52.
[6] Robert Martin, *Clean code. Creation, analysis and refactoring*, 2021.
[7] Ryndin, Nikita and Sapegin, Sergey, "Component design of the complex software systems, based on solutions' multivariant synthesis," *International Journal of Engineering Trends and Technology*, vol. 69, pp. 280–286, 12 2021.
[8] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proceedings of the 8th working conference on mining software repositories*, 2011, pp. 73–82.
[9] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.
[10] Gulyakina N. A., Golenkov V. V., "Graphic-dynamic models of parallel knowledge processing: principles of construction, implementation and design," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, Golenkov V. V., Ed. BSUIR, Minsk, 2012, pp. 23–52.
[11] Brooks F., *Mythical man-month, or How software systems are created*. SPb.: Symbol-Plus, 2021.
[12] G. Sellitto, E. Iannone, Z. Codabux, V. Lenarduzzi, A. De Lucia, F. Palomba, and F. Ferrucci, "Toward understanding the impact of refactoring on program comprehension," in *29th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2022, pp. 1–12.
[13] M. Di Penta, G. Bavota, and F. Zampetti, "On the relationship between refactoring actions and bugs: a differentiated replication," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 556–567.
[14] R. Turner, "Programming languages as technical artifacts," *Philosophy & technology*, vol. 27, no. 3, pp. 377–397, 2014.
[15] R. d. Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. Schmerl, D. Weyns, L. Baresi, N. Bencomo *et al.*, "Software engineering for self-adaptive systems: Research challenges in the provision of assurances," *Software Engineering for Self-Adaptive Systems III. Assurances*, pp. 3–30, 2017.
[16] R. Turner, "Computational artifacts," in *Computational artifacts*. Springer, 2018, pp. 25–29.
[17] Golenkov, V. V., "Methodological problems of the current state of works in the field of artificial intelligence," *Open Semantic Technologies for Intelligent Systems = Open Semantic Technologies for Intelligent Systems (OSTIS-2021): collection of scientific papers / Belarusian State University of Informatics and Radioelectronics*, pp. 17–24, 2021.
[18] R. Turner and A. H. Eden, *Towards a programming language ontology*. na, 2007.
[19] C. Olteanu, "Programming, mathematical reasoning and sense-making," *International Journal of Mathematical Education in Science and Technology*, vol. 53, no. 8, pp. 2046–2064, 2022.
[20] F. W. Neiva, J. M. N. David, R. Braga, and F. Campos, "Towards pragmatic interoperability to support collaboration: A systematic review and mapping of the literature," *Information and Software Technology*, vol. 72, pp. 137–150, 2016.
[21] O. Chaparro, G. Bavota, A. Marcus, and M. Di Penta, "On the impact of refactoring operations on code quality metrics," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 456–460.
[22] N. N. Skeeter, N. V. Ketko, A. B. Simonov, A. G. Gagarin, and I. A. Tislenkova, "Artificial intelligence: Problems and prospects of development," in *13th International Scientific and Practical Conference-Artificial Intelligence Anthropogenic nature Vs. Social Origin*. Springer, 2020, pp. 306–318.
[23] Golenkov V.V., Gulyakina N.A., Davydenko I.T., Shunkevich D. V., Eremeev A.P., "Ontological design of hybrid semantically compatible intelligent systems based on the semantic representation of knowledge," in *Ontologiya proyektirovaniya*, Golenkov V.V., Ed. Russian Federation, Samara: Samara National Research University named after Academician S.P. Korolev, 2019, pp. 132–148.
[24] T. S. Dillon, E. Chang, and P. Wongthongtham, "Ontology-based software engineering-software engineering 2.0," in *19th Australian Conference on Software Engineering (ASWEC 2008)*. IEEE, 2008, pp. 13–23.

[25] D. C. Sales, L. B. Becker, and C. Koliver, "The systems architecture ontology (sao): an ontology-based design method for cyber–physical systems," *Applied Computing and Informatics*, 2022.
[26] S. Elnagar, V. Yoon, and M. A. Thomas, "An automatic ontology generation framework with an organizational perspective," *arXiv preprint arXiv:2201.05910*, 2022.
[27] A. H. Eden and R. Turner, "Problems in the ontology of computer programs," *Applied Ontology*, vol. 2, no. 1, pp. 13–36, 2007.
[28] P. Lando, A. Lapujade, G. Kassel, and F. Fürst, "Towards a general ontology of computer programs," in *International Conference on Software and Data Technologies*, vol. 2. SCITEPRESS, 2007, pp. 163–170.
[29] ——, "An ontological investigation in the field of computer programs," in *Software and Data Technologies*. Springer, 2007, pp. 371–383.
[30] M. J. Jacobs, "A software development project ontology," Master's thesis, University of Twente, 2022.
[31] E. Tin, V. Akman, and M. Ersan, "Towards situation-oriented programming languages," *ACM Sigplan Notices*, vol. 30, no. 1, pp. 27–36, 1995.
[32] H. Schiitze, "The prosit language v0. 4," *Manuscript, Center for the Study of Language and Information, Stanford University, Stanford, CA*, 1991.
[33] A. Black, "An approach to computational situation semantics," Ph.D. dissertation, PhD thesis, Department of Artificial Intelligence, University of Edinburgh . . . , 1993.
[34] W. J. Rapaport, "Syntax, semantics, and computer programs," *Philosophy & Technology*, vol. 33, no. 2, pp. 309–321, 2020.
[35] J. Grimmelmann, "Programming languages and law: A research agenda," *arXiv preprint arXiv:2206.14879*, 2022.
[36] Tetlow, Philip and Garg, Dinesh and Chase, Leigh and Mattingley-Scott, Mark and Bronn, Nicholas and Naidoo, Kugendran and Reinert, Emil, "Towards a semantic information theory (introducing quantum corollas)," 2022.
[37] Dijkstra E., *Programming Discipline*. M.: Mir, 1978.
[38] Reinhard Diestel, *Graph Theory*. Hamburg, Germany: Universität Hamburg, 2017.
[39] Kuznecov, O. P., *Diskretnaya matematika dlya inzhenera: Uchebnik dlya vuzov [Discrete Mathematics for an Engineer: A Textbook for High Schools]*. Moscow: Lan', 2009.
[40] Golenkov, V. V., Gulyakina, N. A., Shunkevich, D. V., *Open technology for ontological design, production and operation of semantically compatible hybrid intelligent computer systems*, Golenkov V.V., Ed. Minsk: Bestprint, 2021.
[41] X. Zhong, E. Cambria, and A. Hussain, "Does semantics aid syntax? an empirical study on named entity recognition and classification," *Neural Computing and Applications*, vol. 34, no. 11, pp. 8373–8384, 2022.
[42] T.-D. Bradley, J. Terilla, and Y. Vlassopoulos, "An enriched category theory of language: from syntax to semantics," *La Matematica*, pp. 1–30, 2022.
[43] F. Zhou, Y. Li, X. Zhang, Q. Wu, X. Lei, and R. Q. Hu, "Cognitive semantic communication systems driven by knowledge graph," *arXiv preprint arXiv:2202.11958*, 2022.
[44] Kasyanov, V. N., Evstigneev, V. A., "Graphs in programming: processing, visualization and application," *BHV–St. Petersburg*, p. 1104, 2003.
[45] Petrov, C. V., "Graphic grammars and automata (overview)," *Automation and telemechanics*, pp. 116–136, 1978.
[46] N. Sales and J. Efson, "An explainable semantic parser for end-user development," Ph.D. dissertation, Universität Passau, 2022.
[47] Ford, Brian and Schiano-Phan, Rosa and Vallejo, Juan, *Component Design*, 11 2019, pp. 160–174.
[48] V. Kabilan, "Ontology for information systems (o4is) design methodology," 2007.
[49] Y. I. Molorodov, "Development of information system based on ontological design patterns," in *CEUR Workshop Proceedings*, 2019, pp. 26–30.
[50] Tuzov, V. A., "On the formalization of the task concept," *M: Science*, pp. 73–83, 1986.
[51] Pospelov, D. A., "Situational management. theory and practice," *M: Science*, p. 288, 2021.
[52] K. Lu, Q. Zhou, R. Li, Z. Zhao, X. Chen, J. Wu, and H. Zhang, "Rethinking modern communication from semantic coding to semantic communication," *IEEE Wireless Communications*, 2022.
[53] Scott, M. L., *Programming Language Pragmatics*. Morgan Kaufmann publications, 2006.
[54] Scott, D., *Lattice Theory, Data Types and Formal Semantics, Formal Semantics of Programming Languages*. Prentice-Hall, Englewood Cliffs, NJ, 1972.
[55] R. Lil, H. Zhu, and R. Banach, "Denotational and algebraic semantics for cyber-physical systems," in *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2022, pp. 123–132.
[56] Orlov, S.A., "Theory and practice of programming languages," *St. Petersburg: Peter*, 2013.
[57] Ben-Ari M., *Programming languages. Practical Benchmarking*. M.: Mir, 2000.
[58] Gulyakina N.A., Pivovarchik O.V., Lazurkin D.A., "Languages and programming technology focused on the processing of semantic networks," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*. BSUIR, Minsk, 2012, pp. 222–228.
[59] D. Shunkevich, D. Koronchik, "Ontological approach to the development of a software model of a semantic computer based on the traditional computer architecture," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*. BSUIR, Minsk, 2021, pp. 75–92.
[60] Donald Knuth, *The art of programming. Volume 1. Basic Algorithms*, 2019.

Семантическая теория программ в интеллектуальных компьютерных системах нового поколения

Зотов Н.В.

Несмотря на активное развитие и использование языков программирования, общей теории программ, на основе которой можно было бы проектировать и разрабатывать прикладные системы, на данный момент не существует. В данной работе предлагается единая онтология языков программирования и представления программ на разных языках программирования. Работа показывает особенности представления программ и ключевые моменты процесса их интерпретации.

Received 28.10.2022