Family of external languages of next-generation computer systems, close to the language of the internal semantic representation of knowledge

Alexandra Zhmyrko
Belarusian State University of Informatics and Radioelectronics
Minsk, Belarus
Email: aleksashazh@gmail.com

***Abstract*—In the article, the concepts of external and internal languages of next-generation intelligent computer systems are considered. External languages of knowledge representation within the OSTIS Technology are described, namely SCg-code, SCs-code, SCn-code. For each of the external languages, its syntax and denotational semantics are considered in detail.**

***Keywords*—next-generation intelligent computer system, external language, internal language, OSTIS, SC-code, SCg-code, SCs-code, SCn-code**

I. Introduction

At the current stage of information technologies development, the problem of ensuring semantic interoperability of computer systems and their components is the most important and significant. To solve this problem it is necessary to move from traditional computer systems and modern intelligent computer systems to computer systems based on the semantic representation of information (next-generation intelligent computer systems).

Such systems have a high level of learnability, i.e. the ability to rapidly acquire new and improve already immersed knowledge and skills, while having no limitations on the type of knowledge and skills gained, improved, and shared. The components of such systems have a high degree of compatibility, which virtually eliminates the duplication of engineering solutions and makes it possible to significantly accelerate the development of computer systems based on the semantic representation of information through a constantly expanding library of reusable and compatible components.

Next-generation intelligent computer systems require an internal language to represent information in a meaningful way. By internal language is meant the language used by a system to represent the information stored in its memory [1].

For operation of next-generation intelligent computer systems, except for a method of abstract internal representation of knowledge bases, methods of external representation of abstract texts convenient for users and used at registration of initial texts of knowledge bases of the specified intelligent computer systems and initial texts of fragments of these bases, as well as used for display of various fragments of knowledge bases on user request, are required [2].

All basic external formal languages are variants for the external representation of the texts of the internal language of the system. Such languages are universal and therefore semantically equivalent.

For any language, syntactic rules (rules for constructing information constructions of such a language) and semantic rules (denotational semantics – rules for relating to those entities and configurations of entities that are described (reflected) by the specified sign constructions) must be specified.

A next-generation intelligent system must be able to visualise certain information in different ways. Each visualisation option requires the design and development of visualisation languages and tools to translate these languages from the system internal representation language into an external language. At the moment, the **lack of a universal mechanism for describing** external languages, translators for them, and their "seamless" integration into the system remains problematic.

In the article, a family of external languages of next-generation intelligent computer systems close to the language of internal knowledge representation on the example of ostis-systems is considered. For each of the external languages, its syntax, denotational semantics, and hierarchical family of semantically equivalent sub-languages are described in detail.

II. State of art

Knowledge representation languages are frequently difficult to understand, particularly for those who is not trained in formal logic. They are in common used to describe domains ranging from biology to finance. These languages are typically used by both computer scientists and domain experts [3].

It is often said that a picture is worth a thousand words. That is true of sketches, diagrams, and graphs

used in various fields of knowledge. Conceptual maps are widely used in education to represent and clarify complex relations between concepts. Flowcharts serve as graphical representations of procedural knowledge or algorithms. Decision trees are another form of representation used in various fields, particularly in decision-making or expert systems.

All these representation methods are useful at an informal level, as thinking aids and tools for the communication of ideas, but they have limitations. One is the imprecise meaning of the links in the model. Non-typed arrows can mean many things, sometimes within the same graph. Another problem is the ambiguity around the type of entities. Objects, actions on objects, and propositions of properties about them are all mixed-up, which make graph interpretation a fuzzy and risky business.

Another difficulty is to combine more than one representation in the same model. For example, concepts used in procedural flowcharts as entry, intermediate, or terminal objects could be given a more precise meaning by developing them in conceptual maps as sub-models of the procedure. The same is true of procedures represented in conceptual models that could be developed as procedural sub-models described by flowcharts, combined or not with decision trees. In software engineering, many graphic representation formalisms have been or are used such as EntityRelationship models [4], Conceptual Graphs [5], Object modelling technique (OMT) [6], KADS [7], or the Unified Modeling Language (UML) [8]. These representation systems have been built for the analysis and architectural design of complex information systems. The most recent ones require the usage of up to eight different kinds of model so the connections between them become rapidly hard to follow without considerable expertise.

Graphic representation system should be both simple enough to be used by educational specialists who are not computer scientists in general, be general and powerful enough to represent the components of computer-based educational environments and their relations.

Graphic. The benefits of graphical cognitive modelling have been eloquently summarized by Ausubel [9], Dansereau [10], and Jonassen [11]. Graphs illustrate relations among components of complex phenomena. They uncover the complexity of actor interactions. They facilitate the communication about the reality studied. They favour the global comprehension of studied phenomena. They help grasp the structure of related ideas by minimizing the usage of ambiguous natural language texts. As an example, entity-relation graphs reduce ambiguity compared to a natural language description but some remain on the interpretation of the terms written on the connections or nodes. Ambiguity can be reduced further by the usage of standardized typed objects and typed connections.

User-friendliness. Not all graphic modeling languages are user-friendly. A good counter-example is UML. The large number models and symbols require considerable expertise for the interpretation and construction of the model of a system. Furthermore, each type of model captures a different viewpoint on the information, and it is impossible to mix them in the same graph to provide a global view of a subject domain. The representation system must be easy to use without technical or scientific mastery after a short period of initiation. Dansereau and Holley [12] have studied experimentally the usage of different sets of graphic symbols by learners. Their results show that typed connections are preferred by the majority of learner, as long as there are neither too few nor too many types of connections and they are clearly differentiated with well-defined meanings.

General. Generality means that the representation language should have the capacity to represent, with a relatively small number of objects and connection categories, knowledge in very different subject domains, at various levels of granularity and precision. It should be possible to represent simple models such as a multiplication table, up to complex models such as multi-actor workflows, rule-based systems, methods, and theories. It should also be possible to offer equivalent representations to commonly used graphs such as conceptual maps, semantic networks, flowcharts, decision trees, or cause/effect diagrams.

Formalizable. The graphic language should be upward compatible from informal graphs, up to semiformal and totally unambiguous formal models. At the informal level, an integrated representation framework facilitates thought organization and communication between humans about the knowledge as the graphic representation model evolves. Here, the process is more important than the result. At the other hand, the graphic language offers more constrained elements to produced totally unambiguous descriptions that can be exported to set of symbols, such as an XML file, that can be processed by computer agents. Here, the model is more important than the process.

Declarative. Graphic language can be procedural or declarative. Procedural graphic languages have been built in the past, extending flowcharts to promote graphical programming that produces code directly. However, declarative language is, firstly, easier for a human to declare the components of their knowledge than to describe also the way it should be processed. In expert systems, for example, the execution instructions are not wired in the program but externalized and made visible in a knowledge base on which a general inference engine proceeds. Secondly, the same model can be used for many different applications not necessarily the one for which the processing has been planned in a procedural program. This is done by querying the model using an inference engine, in a Prolog-like manner. Thirdly, the processing knowledge itself can be given

declaratively, so that higher order metaknowledge can be also singled-out. This idea is similar to structural analysis [13] and is exactly the way we should see the relation between generic skills and specific domain knowledge in a competency, as meta-knowledge given declaratively, applied to domain knowledge. For example, rules for diagnosing a component-based system applied to models describing a car, a software, or a learning environment provide a good way to represent generic skills and competencies.

Standardized. Standardization is an important property to enlarge knowledge communication and use between humans and/or software agents. At the informal level, each model constructed by a human must be interpretable by another human.

Computable. Computability is a step beyond standardization. The graphic model can not only receive a non-ambiguous formal representation that can be processed by computer agents, but this formal representation is complete (all conclusions are guaranteed to be computable) and decidable (all computations will finish in finite time) [14].

Thus, knowledge representation languages in next-generation intelligent computer systems must comply with the above properties.

III. Proposed approach

To solve the problem of integrating new external languages of knowledge representation into the system, it is proposed to describe external languages on the basis of ontologies. As already mentioned, each language is defined by its syntax and denotational semantics, which can be written in an ontological way, which will allow universalising and docking these languages with each other, creating tools for visualising and understanding the languages, making them more universal.

The OSTIS Technology, a next-generation technology for intelligent computer system design, is proposed as a tool to implement the specified approach.

The advantages of the OSTIS Technology:

- at the heart of the OSTIS Technology, there is an *SC-code*, which allows any information to be represented in a unified (same) way, making the proposed approach universal and suitable for any class of intelligent system;
- the OSTIS Technology and the *SC-code* in particular can be easily integrated with any modern technology, allowing the proposed approach to be applied to a large number of already developed intelligent systems;
- the *SC-code* allows storing and describing in the knowledge base of the *ostis-system* any external (foreign) information in relation to the *SC-code* in the form of internal ostis-system files. Thus, the knowledge base of the training subsystem can contain explicitly fragments of already existing documentation for the system, represented in any form;
- the OSTIS Technology has already developed models for ostis-system knowledge bases, ostis-system problem solvers, and ostis-system user interfaces, assuming their complete description in the system knowledge base. Thus, for ostis-systems, the proposed approach to training end-users and developers is much easier to implement and provides additional benefits;
- one of the main principles of the OSTIS Technology is to ensure the flexibility (modifiability) of the systems developed on its basis. Thus, the usage of the OSTIS Technology will enable the evolution of the intelligent learning subsystem itself [2].

The systems developed on the basis of the OSTIS Technology are called *ostis-systems*. The universality of the *SC-code* is ensured by the fact that the elements of the *SC-code* texts can be signs of described entities of any kind, including connection signs between the described entities and/or their signs. Accordingly, the texts of the *SC-code* are graph structures of an extended form, in which the characters of the described connections can connect not only the vertices (nodes) of the graph structure but also the characters of other connections.

The *SC-code* is an abstract language, i.e. a language for which the way, in which the characters (syntactically elementary fragments) that make up the texts of this language are represented, is not specified but only the alphabet of these characters, i.e. the family of character classes considered syntactically equivalent to each other, is specified.

Each abstract language can be assigned a whole family of real languages providing isomorphic real representation of texts of the specified abstract language by specifying ways of representation (representation, coding) of symbols included in these texts, as well as by specifying rules for establishing syntactic equivalence of these symbols. Obviously, in all other respects, the syntax and denotational semantics of the mentioned real languages completely coincides with the syntax and denotational semantics of the corresponding abstract language.

Every intelligent system operates with a knowledge base in an internal language, and the dialog takes place as an exchange of messages between the user and the system. For such a dialog to take place, a fragment of the knowledge base must be displayed into an external form. Such a form can be either universal or specialized.

Within the *OSTIS Technology*, three universal external knowledge representation languages are proposed:

- the **SCg-code** – one possible way of visually representing *SC-texts*. The basic principle behind the *SCg-code* is that each *sc-element* is matched with an *sc.g-element* (graphical representation);

- the **SCs-code** – string (linear) representation of the *SC-code*, designed to represent sc-graphs (texts of *SC-code*) as sequences of characters;
- the **SCn-code** – string non-linear variant for representation of the *SC-code*. The *SCn-code* is intended to represent *sc-graphs* as formatted sequences of characters according to predefined rules, within which basic hypermedia such as graphical images can be used, as well as means of navigation between parts of *sc.n-texts* [15].

Each of these languages meets the requirements for universal languages of knowledge representation and allows the user to choose the most convenient variant of visual representation of any subject domain. In addition, each of these languages has the unique property of being able to be described **in the same language**, being able to **be translated from one to the other**. Thus, it is proposed to use *SCg-code, SCs-code, and SCn-code* as knowledge representation languages, whose syntax and denotational semantics will be considered within this article.

IV. Internal language of the *ostis-system* – an SC-code

SCg-code, SCs-code, and SCn-code are sub-languages of the *SC-code*, which define the syntactic, semantic, and functional principles of memory organisation in next-generation computers focused on the implementation of next-generation intelligent computer systems.

The *SC-code* texts (sc-texts) are unified semantic networks with a basic set-theoretic interpretation. The elements of such semantic networks are called *sc-elements* (*sc-nodes* and *sc-connectors*, which in turn can be *sc-arcs* or *sc-rules*, depending on their orientation). The *Alphabet of the SC-code* consists of five basic elements, on the basis of which SC-code constructions of any complexity are built, including the introduction of more specific types of sc-elements (e.g. new concepts). A detailed description of the *SC-code* can be found in the standard [16].

The signs (designations) of all entities described in *sc-texts* (SC-code texts) are represented as syntactically elementary (atomic) fragments of *sc-texts* and therefore have no internal structure, not consisting of simpler text fragments, such as names (terms), which represent signs of described entities in familiar languages and consist of letters.

Names (terms), natural language texts, and other information constructions which are not *sc-texts* can be included in *sc-text* but only as files described (specified) by *sc-texts*. Thus, a knowledge base of an intelligent computer system based on the *SC-code* can include names (terms) denoting some describable entities and represented by corresponding files.

Each *sc-element* will be called an internal designator of some entity, and the name of this entity represented by the corresponding file will be called an external identifier (external designator) of this entity. Each named (identifiable) sc-element is connected by an arc of membership to the "to be an external identifier*" relation with a node whose content is an identifier file (in particular, a name) denoting the same entity as the above sc-element. The external identifier can be a name (term) but also a hieroglyph, a pictogram, a spoken name, a gesture. It should be emphasized that external identifiers of described entities in an intelligent computer system based on the *SC-code* are used only:

- to analyse information coming into this system from various sources and to input (understand and immerse) this information into the knowledge base;
- to synthesise different messages addressed to different subjects (including users).

V. Identification of sc-elements

External *sc-element* identifiers (or, for short, *sc-identifiers*) are necessary for the *ostis-system* to exchange information with other *ostis-systems* or with its users. In order to represent its knowledge base, to solve various problems related to analysis of the current state and evolution of its knowledge base, problems related to analysis of the current state (current situations) of the environment, making appropriate decisions (purposes), and organising appropriate actions to implement the decisions made (to achieve the purposes), the *ostis- system* does not need any external identifiers (in particular names) corresponding to *sc-elements*.

However, in order to understand messages received from other subjects (which for the *ostis-system* means to construct the *sc-text* semantically equivalent to the received message) and to analyze messages transmitted to other subjects (which for the *ostis-system* means synthesizing an external text that is semantically equivalent to a given *sc-text* and meeting some additional requirements, such as an emotional one). The *ostis-system* needs to know how characters that are synonymous with sc-elements which are or could be stored in the knowledge base of the *ostis-system* are represented in the message being received or transmitted.

The external identifiers of *sc-elements* are most often the names (terms) of the corresponding (denoted) entities, represented by single words or phrases in various natural languages, however, hieroglyphs, conventions, pictograms can also be used.

In general, an *sc-element* can correspond to several synonymous names in different natural languages. Moreover, an *sc-element* can correspond to several synonymous names in the same natural language. In this case, one of these names is declared as the main external identifier for the corresponding *sc-element* and the corresponding natural language. The main requirement for such external identifiers is that there are no synonyms as well as homonyms within the set of basic external identifiers of *sc-elements* for each natural language.

Each external *sc-element* identifier used by the *ostis-system* can be described (represented) in its memory as an internal *ostis-system* file, i.e. as an electronic image of all possible occurrences of this *external identifier* in all possible external texts of the corresponding external language. In some cases, an explicit representation in memory is not required, e.g. in the case of *non-translatable sc-identifiers*.

Next, let us consider the external universal languages of knowledge representation.

VI. SCg-code. Alphabet of the SCg-code and denotational semantics

An *SCg-code* is a way of visualising *sc-texts* (SC-code information constructions) as drawings of these abstract structures. We emphasize that an abstract graph structure and its drawing (graphical representation) are not the same thing even if they are isomorphic to each other.

We consider the *SCg-code* as an combination of the *SCg-code Core*, which provides an isomorphic graphical representation of any *sc-text* and several extensions to this core that provide increased compactness and "readability" of *SCg-code (sc.g-texts)* texts.

The main purpose of the *SCg-code* is to have a clear syntactic graphic representation of *sc.g-elements*, allowing the classes of *sc.g-elements* to be easily identified and distinguished, such as:

- *sc.g-constants* (signs of constant entities) and sc.g-variables (images of variables whose values are the corresponding sc-elements);
- *sc.g-variables* whose values are sc-constants and sc.g-variables whose values are sc-variables;
- signs of permanent (stable) entities and signs of temporal (unstable, temporary existing, situational) entities;
- *sc.g-connectors* (binary characters) and *sc.g-elements* that are not sc.g-connectors;
- *non-oriented sc.g connectors* (sc.g edges) and oriented (sc.g arcs);
- *sc.g-arcs of membership* and sc.g-arcs that are not such;
- *sc.g-arc* of positive membership, negative membership, and fuzzy membership.

Figure 1 is the element list for the Alphabet of the SCg-code.

This list is created in the form of *sc.g-text* and is a representation for examples of all put types of *sc.g-elements* (one example of each type). At the same time, the specified examples of *sc.g-elements* are divided into five groups (SCg-text. Alphabet of the SCg-code). The first group (top row) includes *sc.g-element* for which the constancy and consistency of the entities they denote requires further specification. The remaining four groups of *sc.g-elements* are similar to each other and include, respectively:

- signs of constant permanent entities;
- signs of constant temporal entities;
- images of sc-variables whose values or whose value values (in case the values of the variables are variables) are the signs of constant permanent entities;
- images of sc-variables whose values or whose value values (in case the values of the variables are variables) are the signs of constant temporal entities.

A special point of the *SCg-code* is the representation of *sc-elements*, which are designations of the membership pair* by explicitly using this semantically distinguishable class of *sc-elements*. This *sc.g-element* is used when we need to represent an *sc-arc* that is known to be a designation of the membership pair*, but it is not known whether it is constant or variable, permanent or temporal, positive, negative, or fuzzy.

In addition to the *sc.g-elements* listed in Figure 1, the Alphabet of the SCg-code also includes the following *sc.g-elements*:

- external sc-element identifiers that are identical (attributed) to the corresponding *sc.g-elements*;
- sc.g-contours, each of which is a sign of some sc-text (a structure consisting of sc-elements). Each such sc-text can be:
  - either a constant permanent structure;
  - a constant temporal structure (situation);
  - or a variable structure whose values are permanent structures of an isomorphic configuration;
  - or a variable structure whose values are temporal structures (situations) of an isomorphic configuration.
- enlarged sc.g-frames that are image limiters for the various files stored in the *ostis-system* memory;
- sc.g-buses, which are designations of the same entities as their incident *sc.g-elements*.

Let us note also that, in addition to all the above elements of the Alphabet of the SCg-code, each of which has quite specific denotational semantics, a number of "smaller" syntactic objects need to be introduced to formalise the SCg-code syntax, e.g:

- incidence points of sc.g-connectors with sc.g-nodes, with other sc.g-connectors, with sc.g-contours, with sc.g-frames;
- sc.g-bus incidence points;
- salient points of linear *sc.g-elements* (sc.g-connectors, sc.g-contours, sc.g-frames, sc.g-buses).

Within the SCg-code, the SCg-code Core and its extensions are allocated. The Alphabet of SCg-code Core is an alphabet of *sc.g-elements* graphically represented by sc-elements. The Alphabet of the SCg-code Core is mutually unambiguous with the Alphabet of the SC-code.

The denotational semantics of the SCg-code Core correspond to the denotational semantics of the SC-code. This is demonstrated in Figure 2.

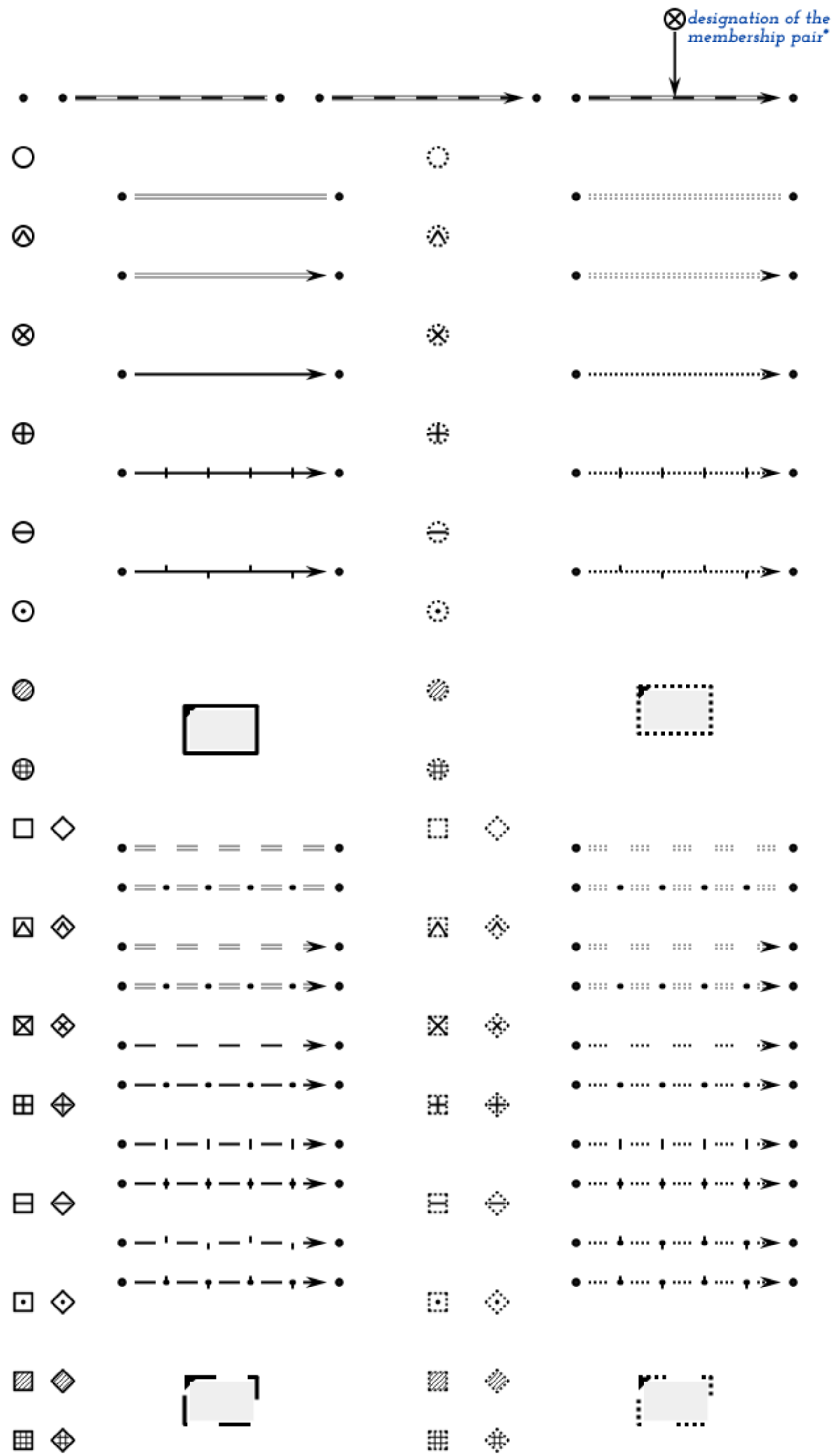

Figure 1. Elements of the Alphabet of the SCg-code

Alphabet of the SC-code	Alphabet of the SC-code Core	sc.g-element image
sc-node of a common type	sc.g-node of a common type	•
sc-edge of a common type	sc.g-edge of a common type	
sc-arc of a common type	sc.g-arcof a common type	
basic sc-arc	basic sc.g-arc	
internal ostis-system file	content sc.g-node	

Figure 2. Denotational semantics of the SCg-code Core

The Alphabet of the SCg-code Core is represented by the following elements:

- an *sc.g-node of a common type* – an *sc.g-element* which is a graphical representation of the sc-node of a common type. All sc-nodes, which are not file signs, in the text (construction) of the SCg-code Core, are represented as small black circles of the same diameter, which we denote by 'd' and the exact value of which depends on the scale of sc.g-text;
- an *sc.g-edge of a common type* – an *sc.g-element*, which is a graphical representation of the sc-edge of a common type. Each sc-edge in the SCg-code Core is represented as a wide line with alternating solid-filled and non-filled fragments that have no self-intersections and an overall weight of about 0.7 d;
- an *sc.g-arc of a common type* – an *sc.g-element*, which is a graphical representation of the sc-arc of a common type. Each sc-arc in the SCg-code Core is represented as a wide line with alternating solid-filled and non-filled fragments, with no self-intersections, having an overall weight of about 0.7 d and having an arrow at one end of this line;
- a *basic sc.g-arc* – an *sc.g-element*, which is a graphical representation of the basic sc-arc. Each underlying sc-arc in the SCg-code Core is represented as an arbitrary shaped line without self-intersections, having a weight of 0.4 d and an arrow at one of its ends;
- an *internal ostis-file* – an *sc-node*, which is the sign of the internal *ostis-system* file;
- an *sc.g-node with contents* – an *sc.g-node*, which is the sign of the internal *ostis-system* file, an *sc.g-frame*;
- an *sc.g-frame* is always a rectangle, the maximum size of which is not limited, but the minimum size is fixed and corresponds to the *sc.g-frame*, inside which the file it designated is not displayed. Each sc-node in the *sc-text* that has contents is represented as an arbitrarily sized rectangle with a line weight of 0.6 d in the *SCg-code* Core. Inside this rectangle, it is possible to see the file indicated by the depicted *sc-node*. If there is no need to represent the file itself in the text, the *sc-node* denoting such a file is represented in *sc.g-text* as a rectangle with sides 2d vertically and 4d horizontally.

It is difficult to believe at once that such a simple alphabet can be used to build a convenient and versatile graph language. Besides, we should not be alarmed by simplicity of the alphabet because mankind has a great experience of coding, storing in memory, and transferring through communication channels the most various information resources using the alphabet consisting of only two classes of elements – ones and zeros. We are talking about a fundamentally different (graphical) way of encoding information in computer systems, but we try to reduce this encoding to a simple enough alphabet at least in order not to artificially complicate the problem of creating next-generation computers based on information encoding method. Extensions of the *SCg-code* Core will be considered as directions of transition from the *SCg-code* Core texts to more compact texts. However, since it leads to complication of the Syntax of the *SCg-code* and, first of all, to expansion of the *Alphabet of the SCg-code*, it is necessary to make such extensions reasonably taking into account frequency of occurrence within knowledge bases of *ostis-systems* of corresponding fragments.

VII. SCs-code. Alphabet of the SCs-code and denotational semantics

An *SCs-code* is a language of linear knowledge representation of *ostis-systems*. A set of linear texts (sc.s-texts), each consisting of sentences (*sc.s-sentences*) separated from each other by a double semicolon (separator of *sc.s-sentences*). In this case, the sc.s-sentence is a sequence of *sc-identifiers* which are the names of the described entities and are separated from each other by different *sc.s-separators* and *sc.s-delimiters*.

The *Alphabet of the SCs-code* is based on modern commonly used character sets, which simplifies the development of tools for working with *sc.s-texts* using modern technologies.

The *sc.s-texts*, as well as the texts of any other languages which are variants of the external representation of the *SC-code* texts, can include various files, including natural language files or even files containing other sc.s-texts. In general, such files can use a variety of characters,

so we will assume that these characters are not included in the *Alphabet of the SCs-code*.

The alphabet of symbols used in *sc.s-separators* consists of: space, semicolon, colon, round marker, and equality sign.

The alphabet of symbols used in *sc.s-separators* displaying the incidence relation of sc-elements consists of: “<”, “>”, “|”, “-”.

The basic alphabet of characters used in *sc.s-connectors* consists of: “∼”, underscore sign, equality sign, colon, ‘<”, “>”, “|”, “-”, “/”.

The extended alphabet of symbols used in *sc.s-connectors* consists of “∈”, “∋”, “⊆”, “⊇”, “⊂”, “⊃”, “≤”, “≥”, “⇐”, “⇒”, “←”, “→”, “⇔”.

Both in the *Basic* and the *Extended* Alphabets of *sc.s-connectors*, the following common features to characterize the type of *sc-connector* being represented are used:

- an underscore as an image feature of the *sc-connectors* variables (one underscore for *sc-connectors* that are primary *sc-variables*, two underscores for *sc-connectors* that are secondary *sc-variables (sc-meta-variables)*);
- a vertical line “|” as an image feature of *negative sc-arcs of membership*;
- slash “/” as an image feature of *fuzzy sc-arcs of membership*;
- tilde “∼” as an image sign of *temporal sc-arcs of membership*.

To simplify the process of developing knowledge base source texts using the *SCs-code* and creating corresponding tools, two character alphabets are introduced. The basic alphabet of characters used in *sc.s-connectors* includes only the characters included in the portable character set and available on a standard modern keyboard. Thus, to develop the source code of knowledge bases using only the Basic alphabet of symbols used in *sc.s-connector*, a normal text editor is sufficient. The extended alphabet of characters used in *sc.s-connectors* also includes additional characters that make *sc.s-texts* (and *sc.n-texts*) more readable and clear. To visualize and develop sc.s-texts using the extended alphabet, specialized tools are required.

The alphabet of symbols used in *sc.s-delimiters* consists of: “(”, “)”, “*”.

The alphabet of symbols used in ambiguous *sc.s-images of sc-nodes* consists of: “{”, “}”, “-”, “!”, “ [”, “] .

Important elements of the *SCs-code* are the *sc.s-separator* and the *sc.s-delimiter*.

An *sc.s-separator* is a separator used in *sc.s-texts*. The *sc.s-separator* splits into:

- an *sc.s-separator* used to structure *sc.s-sentences*.
  - it separates the *sc-identifier of a binary relation* and the second component of one of its connectives, in case the specified binary relation and its connective are connected by a *constant sc-arc of membership*. It is represented as a colon.
  - Separates the *sc-identifier of a binary relation* and the second component of one of its connectives, in case the specified binary relation and its connective are connected by a *variable sc-arc of membership*. It is represented as a double colon.
- *sc.s-separator* of *sc.s-sentences* is represented as a double semicolon.

sc.s-delimiter is represented as: (![(∗]!∪![∗)]!)

The parentheses with an asterisk limit attached *sc.s-sentences*, which, in turn, may have other attached *sc.s-sentences* in their structure.

There is also an *sc.s-connector*. The typology of *sc.s-connectors* is fully consistent with that of *sc.g-connectors* and even more so with *sc-connectors*, since it takes into account the well-established tradition of representing the connectives of a number of specific relations. The following *sc.s-connectors* are distinguished:

- an *oriented sc.s-connector*,
- a *non-oriented sc.s-connector*;
- an *sc.s-connector*, corresponding to the *sc.g-arc of membership*,
- an *sc.s-connector* corresponding to a *sc.g-connector* that is not an *sc.g-arc*.

The set of *sc-elements* has a binary oriented sc-element incidence relation, as well as a subset of this relation – the incidence relation of incoming *sc-arcs*, each pair of which relates the sc-arc to the sc-element it is a part of. In the *SC-code*, *sc-connectors* can connect not only an *sc-node* with *sc-nodes* but also an *sc-node* with an *sc-connector* and even an *sc-connector* with an *sc-connector*. In the latter case, specifying the incidence of sc-connectors, it is necessary to specify which of them is connecting and which is connectable. Therefore, the incidence relation specified on the set of sc-elements is oriented. The first component of the pair of this relation is the connecting *sc-connector* and the second component is the connecting *sc-element*. Obviously, the connecting *sc-element* is always an *sc-connector* and the *sc-node* can only be connectable.

The *sc.s-separator* displaying the incidence relation of *sc-elements* is divided into:

- incidence sign of the “right” *sc-connector* – the incidence sign of the *sc-connector* whose *sc-identifier* is on the right, represented as “⊢”;
- incidence sign of the “left” *sc-connector* – the incidence sign of the *sc-identifier* whose *sc-identifier* is on the left, represented as “⊣”;
- incidence sign of the incoming *sc-arc* on the right is the incidence sign of the *sc-arc*, whose *sc-identifier* is on the right, represented as “| <”;
- incidence sign of the incoming *sc-arc* on the left is the incidence sign of the *sc-arc*, whose *sc-identifier* is on the left, represented as “> |”.

Specified *sc.s-separators* are similar to *sc.s-sentences* in terms of their syntactic structure, but in terms of their denotational semantics, unlike *sc.s-connectors*, they are not representations of corresponding *sc-connectors*.

In Figure 3, an image of *sc.s-connectors* of the Basic and Extended alphabet corresponding to *sc.g-connectors*, which are *sc.g-arc of membership*, is shown.

The equality sign is the *sc.s-separator* of two *sc-identifiers* which identify (name) the same entity and, therefore, are *sc-identifiers** (external unique images) of the same *sc-element*. Most often, one of these two sc-identifiers is a simple *sc-identifier* and the other is an *sc-expression*. Rarely, both of these *sc-identifiers* are *sc-expressions*. And quite rarely, they are both simple sc-identifiers. The latter indicates that both of these *sc-identifiers* are basic *sc-identifiers** of the same *sc-element*. An example:

SC-code = sc.s-text;;

Here, the first *sc-identifier* is a proper name and the second is a common noun.

When translating *sc.s-text* into the *SC-code*, the equality sign may at some stage be matched with an *sc-edge* which belongs to the synonymy* relation of the *sc-elements* identified by the *sc-identifiers* connected by the equality sign. However, in the next step, the specified *sc-edge* is removed, and the *sc-elements* connected by it are patched together. Thus, the *sc-edge* belonging to the synonymy* relation of sc-elements has not only denotational but also operational semantics.

An equality sign with inclusion is an image of an *sc-arc* belonging to an immersion* relation connecting two *sc-nodes* denoting *sc-texts*, the first of which is immersing and the second (in which specified *sc-arc* comes) is immersed, introduced into the first *sc-text*. The *sc-arc* belonging to the immersion* relation is interpreted as a command to immerse one sc-text into the composition of another. When this command is executed, (1) all sc-elements of the immersing *sc-text* become elements belonging to the immersing sc-text, (2) all synonymous *sc-elements* that happen to be part of the immersing *sc-text* are patched together, (3) the *sc-node* denoting the immersing *sc-text*, as well as the specification of this *sc-text* (including the list of all its *sc-elements*), is immersed in the history of the knowledge base evolution together with the specification of the event of immersion of the considered *sc-text* into the knowledge base.

In Figure 4, the *Alphabet of sc.s-connectors* corresponding to *sc.g-connectors* that are not *sc.arcs of membership* is shown.

The minimum semantically coherent fragment of *sc.s-text* is the *sc.s-sentence*; an *sc.s-sentence*, (1) consisting of either two *sc-identifiers* connected by an *sc.s-connector* or three *sc-identifiers* separated by *sc. separators* representing an incidence relation of *sc-elements* and (2) ending with a double semicolon.

It is easy to notice that simple *sc.s-sentences* are essentially the same as RDF triplets, except that a simple *sc.s-sentence* can be "unfolded" using *sc.s-sentence conversion** without changing its meaning, while an RDF-triplet cannot ensure that. This is one of the reasons why, unlike RDF triplets, in simple *sc.s-sentences*, *sc.s-connectors* and *sc.s-separators* displaying the sc-element incidence relation cannot be omitted, since they also show the direction of the relation they display between the *sc.s-elements*.

The operations defined on the set of *sc.s-sentences* can be divided into three groups:

- a group of conversion operations of *sc.s-sentences* consisting of a single operation;
- a group of combination operations of *sc.s-sentence*;
- a group of decomposition operations of *sc.s-sentences* and, in particular, decomposition operations of *sc.s-sentences*.

The list of operations defined on the set of *sc.s-sentences* is as follows:

- Conversion operation of the *sc.s-sentence**. Every *sc.s-sentence* (including the simple *sc.s-sentence*) can be transformed into a semantically equivalent *sc.s-sentence* by a conversion ('reversal') of the chain of *sc.s-sentence* components. Thus, for example, when converting ("unfolding") a simple *sc.s-sentence*
  - its first *sc-identifier* (the first component of this *sc.s-sentence*) becomes the third component of the converted *sc.s-sentence*;
  - its second *sc-identifier* (the third component of the original *sc.s-sentence*) becomes the first component of the "converted" one;
  - the second component of the original *sc.s-sentence* (*sc.s-connector* or sc.s-separator, representing the sc-element incidence relation connecting the above components) remains the second component of the converted *sc.s-sentence*, but it changes direction (" $\ni$ " is replaced by " $\in$ " and vice versa, " $\supset$ " by " $\subset$ " and vice versa, " $\Rightarrow$ " by " $\Leftarrow$ " and vice versa, etc.). We can talk not only about the conversion of *sc.s-sentence* but also about the conversion of *sc.s-connector*, the conversion of sc.s-separator displaying the incidence relation of sc.s-elements.
- The attachment operation of the *sc. s-sentence** is the operation of attaching two *sc.s-sentence* when the last component of the first sentence matches the first component of the second one*. As a result of performing this operation:
  - the first component of the second *sc.s-sentence* is deleted;
  - the rest of the second sentence is surrounded by the sc.s-delimiter of attached sentences " (* " and " *) ". The separator of *sc.s-sentences* " ;; " also

sc-element class	Image of *sc.g-connectors*	Image of *sc.n-connector* in Extended alphabet		Image of *sc.n-connector* in Base alphabet	
constant permanent positive sc-arc of membership		∋	∈	->	<-
constant permanent negative sc-arc of membership		∌	∉	-|>	<|-
constant permanent fuzzy sc-arc of membership		/∋	∈/	-/>	</-
constant temporal positive sc-arc of membership		~∋	∈~	~>	<~
constant temporal negative sc-arc of membership		~∌	∉~	~|>	<|~
constant temporal fuzzy sc-arc of membership		~/∋	∈/~	~/>	</~
variable permanent positive sc-arc of membership		_∋	_∈	_->	<-_
variable permanent negative sc-arc of membership		_∌	∉_	_-|>	<|-_
variable permanent fuzzy sc-arc of membership		_/∋	∈/_	_-/>	</-_
variable temporal positive sc-arc of membership		_~∋	∈~_	_~>	<~_
variable temporal negative sc-arc of membership		_~∌	∉~_	_~|>	<|~_
variable temporal fuzzy sc-arc of membership		_~/∋	∈/~_	_~/>	</~_
metavariable permanent positive sc-arc of membership		__∋	∈__	__->	<-__
metavariable permanent negative sc-arc of membership		__∌	∉__	__-|>	<|-__
metavariable permanent fuzzy sc-arc of membership		__/∋	∈/__	__-/>	</-__
metavariable temporal positive sc-arc of membership		__~∋	∈~__	__~>	<~__
metavariable temporal negative sc-arc of membership		__~∌	∉~__	__~|>	<|~__
metavariable temporal fuzzy sc-arc of membership		__~/∋	∈/~__	__~/>	</~__

Figure 3. An image of *sc.s-connectors* of the Basic and Extended alphabet corresponding to *sc.g-connectors*, which are *sc.g-arc of membership*

Image of *sc-connector* (SCg)	Image of *sc.n-connector* in Extended alphabet		Image of *sc.n-connector* in Base alphabet	
	↔		<>	
	→	←	»	«
	⇔		<=>	
	⇒	⇐	=>	<=
	~⇔		~<=>	
	~⇒	⇐~	~=>	<=~
	_⇔		_<=>	
	_⇒	⇐_	_=>	<=_
	_~⇔		_~<=>	
	_~⇒	⇐~_	_~=>	<=~_
	__⇔		__<=>	
	__⇒	⇐__	__=>	<=__
	__~⇔		__~<=>	
	__~⇒	⇐~__	__~⇒	⇐~__
inclusion*	⊇	⊆		
inclusion*	_⊇	⊆_		
strict inclusion*	⊃	⊂		
strict inclusion*	_⊃	⊂_		

Image of *sc-connector* (SCg)	Image of *sc.n-connector* in Extended alphabet	
order of magnitude*	≥	≤
order of magnitude*	_≥	_≤
strict order of magnitude*	>	<
strict order of magnitude*	_>	<_
external identifier*	:=	
external identifier*	_:=	
synonymy*	=	
plunge*	⊃=	=⊂

Figure 4. The Alphabet of *sc.s-connectors* corresponding to *sc.g-connectors* that are not *sc.arcs of membership*

falls inside the specified delimiter;
- the resulting construction is placed between the last component of the first sentence and the *sc.s-sentence* separator that ended the first sentence;
- the second sentence thus becomes an attached sc.s-sentence.

Similarly, any attached *sc. s-sentence* can be "docked" with other attached *sc.s-sentences*, in general, the level of such nesting is not limited.

- The merge operation of *sc.s-sentences** is the operation of attaching a simple *sc.s-sentence* to an sc.s-sentence where the last *sc.s-connector* is the same as the *sc.s-connector* of the simple *sc.s-sentence* and the sc-identifier preceding that *sc.s-connector* is the same as the first sc-identifier of the simple *sc.s-sentence*
  This operation causes the matching of *sc.s-identifiers* and *sc.s-connector* of the linked *sc.s-sentences* to be "patched" together, and the last *sc.s-identifiers* of the linked *sc.s-sentence* become the last components of the merged *sc.s-sentence*, separated by semicolons. In the same way, any number of simple *sc.s-sentence* can be attached.
- The decomposition operation* of *sc.s-sentences* into simple *sc.s-sentences*
  Every sc.s-sentence can be decomposed into a set of simple *sc.s-sentences*, i.e. represented as a sequence of simple *sc.s-sentences*.
- The decomposition operation of *sc.s-sentences* into simple *sc.s-sentences* with the sc.s-separator representing the incidence relation of the sc-elements*
  Each *sc.s-sentence* (including a simple sc.s-sentence with an *sc.s-connector*) can be represented as a semantically equivalent sequence of simple *sc.s-sentences with sc.s-separator* displaying the incidence relation of sc-elements. This operation uniquely generates a set of *simple sc.s-sentences* of the specified kind.

Obviously, the combination operations of *sc.s-sentences* and the decomposition operations of *sc.s-sentences* are

inverse operations to each other.

From the semantic point of view, the sc.s-sentence is a description of some route in the corresponding sc-text, which is a graph structure of a special kind and whose structure is described (displayed) with *sc.s-sentences*. The specified route is "traversed" by sc-connectors and sc-element incidence relations, if the route passes through incident sc-connectors. The description of the specified route may additionally specify the sets (most often relations) to which the sc-connectors included in the described route belong. In addition, the specified route may have branches at the beginning and/or at the end, where any sc-element is equally incidental to several sc-connectors of the same type, connecting the specified sc-element to some other sc-elements. Thus, each specified branching consists of an unlimited number of branches, each of which consists of one sc-connector and one sc-element connected by it.

A sequence of *sc.s-sentences* separated by double dots forms the sc.s-text. Accordingly, the sc.s-sentence is the minimum sc.s-text.

The meaning of the sc.s-text (as well as the sc.s-text included in the structure) does not depend on the order of *sc.s-sentences* in these sc-texts. That is, rearranging *sc.s-sentences* within such sc.s-texts does not change the meaning of these sc.s-texts (i.e. leads to semantically equivalent sc.s-texts), but greatly affects the human perception (the "readability") of these texts.

Similar to SCg-code, the SCs-code has a sublanguage – the SCs-code Core, which uses a minimal set of syntactic tools but has a semantic power equivalent to the power of SCs-code as a whole.

In the *SCs-code Core*:

- only simple *sc-identifiers* are used, including *sc-identifiers* of external ostis-files (sc-expressions are not used);
- only sc.s-separators are used, displaying the incidence relation of sc-elements, and *sc.s-connectors* displaying a constant permanent positive pair of membership (" $\in$ " and " $\ni$ " in the Extended Alphabet and " $\rightarrow$ *or* " and " $\leftarrow$ " in the Basic Alphabet). Other *sc.s-connector* are not used;
- sc.s-modifiers and, consequently, colons, which are a sign of completion of sc.s-modifiers, are not used;
- only simple *sc.s-sentences*, which, as follows from the above properties of the SCs-code Core, either consist of two simple sc-identifiers connected by a *sc.s-connector* representing a constant permanent positive pair of membership or three simple sc-identifiers separated by sc.s-separators representing an incendence relation of sc-elements are used.

It follows from the above properties of the *SCs-code Core* that in order to represent any sc-text by means of the *SCs-code Core* it is necessary for all sc-elements of this sc-text (except constant permanent positive pairs of membership) to build simple *sc-identifiers* corresponding to them, i.e. it is necessary to name all the specified sc-elements. In turn, the type of each used sc-element (except the constant permanent positive pairs of membership) is specified explicitly by indicating the membership of these elements to the corresponding sc-element classes, including the classes included in the *SC-code Core*.

As it is possible to notice from the above description, the *SCs-code Core* corresponds to the *SCg-code Core*, except that the *SCg-code Core* does not need to name all represented sc-elements, and also in the *SCg-code*, there are graphic images for sc-elements that belong to the corresponding classes of the *SC-code Core*, and this membership need not to be explicitly specified.

Obviously, it is inconvenient and inefficient to use the *SCs-code Core* for writing large fragments of knowledge bases in practice. Nevertheless, from a practical point of view, the *SCs-code Core* can be used, for example, to exchange information with third-party graph representation tools designed to represent information in the form of triplets (e.g., RDF storages). Syntactic extensions to the *SCs-code Core* are needed to enable wider practical usage with next purposes:

- to minimize the number of identifiable (named) sc-elements by using sc-expressions and eliminating the need to identify (name) all sc-elements;
- reducing text by minimizing the number of repetitions of the same sc-identifier by linking *sc.s-sentences*;
- to increase the visibility, "readability" of the sc.s-texts.

Next, let us consider the structured knowledge representation language of *ostis-systems* – an *SCn-code*.

VIII. SCn-code. Alphabet of the SCn-code and denotational semantics

An *SCn-code* is a language for the structured external representation of the *SC-code* texts, which is a syntactic extension of the *SCs-code*, aimed at increasing the clarity and compactness of the *SCs-code* texts.

The *SCn-code* allows switching from linear texts of the *SCs-code* to formatted and actually two-dimensional texts in which there appears a decomposition of the original linear text of the *SCs-code* into lines placed "vertically". In this case, the beginning of all lines of text is fixed and defined by a known and limited set of rules, which makes it possible to use this when formatting sc.n-text (text belonging to the *SCn-code*).

An *SCn-code* is a language of two-dimensional texts. Accordingly, each text of such a language is defined by:

- a set of characters included in it;
- a "horizontal" character order (sequence) relation;
- a "vertical" character order (sequence) relation.

A character that is part of a two-dimensional text can generally have four "adjacent" characters:

- a character to its left within the same line;
- a character to its right within the same line;
- a character located strictly above it in the previous line;
- a character located strictly below it in the next line of text.

Due to the fact that sc.n-texts can include both sc.s-texts and sc.g-texts (delimited by the sc.n-contour), the *SCn-code* can be considered an integrator of different external knowledge representation languages. This makes it possible to compensate the disadvantages of one of the proposed options for external representation of sc-texts (*SCg-code*, *SCs-code*, *SCn-code*) with the advantages of other options when visualizing and developing the knowledge base of the *ostis-system*.

In this case, there is a transition from linearity of sc.s-texts to two-dimensionality of sc.n-texts.

An important feature of the *SCn-code* is the "two-dimensional" nature of its texts. This is manifested in that for each *SCn-code* fragment of text, the value of the indentation from the left edge of the line is essential.

In the *SCn-code* text, unlike the *SCs-code* text, the important thing for each text fragment is not only how this fragment is connected to other fragments "horizontally" (which fragment is to the left or to the right of the same line) but also how it is related to other fragments "vertically" (which fragment is higher on the previous line and which is lower on the next line), which fragment is below on the next line).

So, for example, if in the text of the *SCn-code* some sc-identifier(external sc-element identifier) is placed immediately after the vertical tab line and a certain *sc.s-connector* is placed exactly below it, it means that the specified sc-element is incident to the sc-connector represented by the specified *sc.s-connector*.

In order to provide the exact setting (formulation) of the rules of two-dimensional incidence elements (elementary fragments) of sc.n-texts, the concept of sc.n-text page is introduced, the concept of a line of sc.n-text, and also a special markup is used, which is vertical tab lines, the distance between which is approximately equal to the maximum length of the *sc.s-connector* (usually this distance equals the width of 4-5 characters).

The sc.n-text (text of the *SCn-code*) is a sequence of sentences of the *SCn-code*, each of which is not part of any other sentence in the sequence.

If the sc.n-text is part of some other file that is paginated, such as the publication of some part of a knowledge base, then the sc.n-page is only the part of the page that shows the sc.n-text, while the page of the specified file may be larger due to, for example, white fields on the edges of the page needed for subsequent printing.

The maximum number of characters in sc.n-text lines for each sc.n-text is fixed and is determined by the specific sc.n-text placement option. At the same time, depending on the indentation within a particular sc.n-text sentence, a line of sc.n-text may not start from the left edge of the sc.n-text (but always from some of the vertical markup lines) and have an arbitrary length limited by the right border of the sc.n-page.

A markup line is used to make sc.n-texts easier to read. The 1st markup line borders the left edge of the sc.n-page, the 2nd markup line is located approximately between the 5th and 6th characters of the line, and so on. The distance between the markup lines may vary depending on the font size but always remains the same within a single sc.n-text. The total number of markup lines is limited by the maximum possible width of the sc.n-page in the particular *ostis-system* file containing that sc.n-text.

The Alphabet of the *SCn-code* is the same as the Alphabet of the *SCs-code*. All components of sc.s-texts are also used in sc.n-texts:

- *sc-identifiers*;
- *sc.s-identifiers*;
- modifiers of *sc.s-connectors* with the corresponding delimiters (colons);
- separators used in sc-expressions denoting sc-multiples given by enumeration of elements with corresponding separators (semicolon or round marker);
- round markers in enumerations of sc-element identifiers linked by the same-type sc-connectors with the same-type modifiers to a given sc-element;
- sentence separators (double semicolons) (omitted when converting *sc.s-sentences* to sc.n-sentences);
- delimiters of attached *sc.s-sentences* (omitted when converting *sc.s-sentences* to sc.n-sentences).

However, unlike sc.s-texts in sc.n-texts:

- new kinds of sc-expressions (namely, sc-expressions that have a two-dimensional character) are added;
- a new kind of sentence separators – a blank line – is added;
- the placement of sentences, taking into account the two-dimensional nature of this placement, is changed.

New types of sc-expressions are added to the *SCn-code* compared to the *SCs-code*:

- an sc-expression, which is a two-dimensional *sc.n-text* delimited by an *sc.n-contour* or an *sc.n-frame*. Each *sc.n-contour* is represented conventionally as an opening curly bracket and a closing curly bracket located strictly below it through several lines. Inside these brackets (starting from the vertical markup line where the brackets are located to the right page edge), sc.n-text is placed. The resulting *sc.n-frame* is an image of the structure resulting from the translation of the specified *sc.n-text* into the *SC-code*. Each sc.n-frame is represented in the same way, only instead of curly braces it uses square brackets or square

brackets with an exclamation mark (in the case of a sample file);
- an sc-expression, which is a two-dimensional sc.g-text delimited by an sc.n-contour or an sc.n-frame;
- an sc-expression, which is a two-dimensional graphical representation of an information construct encoded in some *ostis-system* file, delimited by the sc.n-frame. Such an information construction can be a table, a picture, a photograph, a diagram, a graph, and more.

It is easy to notice that an *sc.n-contour* is essentially the two-dimensional equivalent of the sc-expression structure, and an sc.n-frame is the two-dimensional equivalent of the *sc-expression of the inner file* of the *ostis-system* or *sc-expression* denoting the pattern file of the *ostis-system*.

From a formal point of view, an *sc.n-frame* is always a single line of *sc.n-text*. This means that the *sc.n-frame* cannot be syntactically divided into parts within the *sc.n-text* in which it is used and cannot be inserted inside it, for example, with attached *sc.n-sentences* or any other text (unless the *sc.n-frame* contains *sc.n-text*, but in this case specified *sc.n-text* will still be considered as a complete external file and not as a fragment of the surrounding *sc.n-text*).

The *sc.n-sentences* uses a delimiter that is a representation of the structure, which is called an *sc.n-contour*.

The concept of the *sc.n-sentence* is a natural generalization of the concept of the *sc.s-sentence*. Moreover, similarly for *sc.s-sentences*, the concept of concepts are introduced:

- of a simple *sc.n-sentence*;
- of a complex *sc.n-sentence*;
- of an *sc.n-sentence* containing attached *sc.n-sentence*;
- of an *sc.n-sentence* that does not contain any attached *sc.n-sentence*;
- of an attached *sc.n-sentence*;
- of unattached *sc.n-sentence*.

If each unattached *sc.s-sentence* is either the first sentence of the *sc.s-text* or begins after the *sc.s-sentence* separator (double semicolon), then each unattached *sc.n-sentence* starts at the beginning of a new line.

If each attached *sc.s-sentence* starts either after the opening delimiter (opening bracket with an asterisk) or after the separator of the
textitsc.s-sentence, then each attached *sc.n-sentence* starts on a new line under the *sc-identifier* that ends that *sc.n-sentence* (and accordingly, *sc.s-sentence*, respectively) in which this attached *sc.n-sentence* is embedded.

The first *sc-identifier* that is part of the *sc.n-sentence* before the *sc.s-connector* is highlighted in bold italics.

In *sc.n-sentences*, the double semicolon is not used as a sign of completion of these sentences and therefore is not used as a separator for *sc.n-sentences*. Such a separator is an empty line.

The two-dimensionality of the *SCn-code* gives more possibilities (degrees of freedom) for a clear and compact layout of the *sc.n-sentences*.

When the *sc.n-sentence* is drawn up, all the *sc.n-sentences* attached to it are clearly tabulated and attached to the original "vertical" one. The vertical tabulation line specifies the left border of the original (maximum) *sc.n-sentence* or the left border of the *sc.n-sentence* attached vertically.

The left border of the *sc.n-sentence* specifies the start of the first *sc.n-sentence* that is part of this *sc.n-sentence* and the start of the *sc.s-connector* that is incident to the specified *sc.s-identifier* and is placed strictly below this *sc-identifier*. The distance between the vertical tab lines is fixed and approximately equal to the maximum length of the *sc.s-connector*.

In contrast to *sc.s-texts*, in *sc.n-texts*, an *sc.s-connector* can be incident to the preceding *sc-identifier* (either simple one or an *sc-expression*) not only "horizontally" but also "vertically". To do this, the *sc.s-connector* is placed strictly below the *sc-identifier* that precedes it.

Also "vertical" *sc-identifier* can be incident to not one but several *sc.s-connectors*, which are consecutively "vertically" placed under the specified *sc-identifier*. This allows within one *sc.n-sentence* representing an arbitrary number of "branches" from each *sc-identifier*, i.e. an arbitrary number of *sc.s-connectors* incident to that *sc-identifier*.

Each *sc-identifier*, including the *sc-expression* delimited by curly or square brackets, must be placed immediately to the right of the vertical marking line if an *sc.s-connector* is placed below it.

Each *sc.s-connector* is highlighted in bold, non-cursive font and, if it is below an incident *sc-identifier*, is placed strictly between the two vertical marking lines, nestled to the left of these two marking lines.

Since in relation to the *SCn-code*, the *SCs-code* is the syntactic core of the language*, the *SCn-code* can be considered as the result of integrating several extensions of the *SCs-code* based on the syntactic transformation rules of *sc.s-texts* and *sc.n-texts*, oriented towards making better usage of those possibilities of visibility and compactness of *sc.n-texts* which are opened by the transition from linearity of *sc.s-texts* to two-dimensional *sc.g-texts*.

The list of operations defined on the set of *sc.n-sentences* is as follows:

- Transformation operation of *sc.s-sentence* to the *sc.n-sentence**
  Every *sc.s-sentence* written linearly ("horizontally") can be transformed into the corresponding two-dimensional *sc.n-sentence*. Let us list the basic rules for transforming *sc.s-sentences* into *sc.n-sentences*
  - The *sc.s-connector* can be placed on the next line below the preceding *sc-identifier*, starting from

the same character of the next line as the specified *sc-identifier*;

- If the *sc-identifier* is moved to the next line, it is continued on the next line with the same indentation from the beginning of the line as the specified *sc-identifier* starts;
- A semicolon-delimited listing of *sc-identifiers* can be carried out not “line by line” but “column by column” by placing each following *sc-identifier* strictly below the preceding one. In this case, the semicolon separator can be replaced by a circle marker placed in front of each *sc-identifier* to be enumerated;
- a closing curly or square bracket may be placed strictly below the corresponding opening bracket;
- The *sc-identifier* in the *sc.n-sentence* can be connected to other *sc-identifiers* via several different *sc.s-connectors*. In this case, each of these *sc.s-connector* is placed strictly below the preceding one but only after the recording of the entire, generally branched, chain of *sc.s-connector* and *sc-identifier* that starts with the preceding *sc.s-connector* is completed. In the *SCs-code*, there is no analogue to such sentences with the unrestricted possibility of describing “branched” connections of *sc-identifiers*. Consequently, if in *sc.s-text*, the *sc-identifier* can be incident to no more than two *sc.s-connectors* (to its left and right), then in *sc.n-text*, sc-identifier can additionally be incident to an unlimited number (not necessarily identical) of *sc.s-connectors* that are placed “vertically” strictly below it.

- Attachment operation of the *sc.n-sentence**
  Some *sc.n-sentence* can be attached to another *sc.n-sentence* if this other *sc.n-sentence* has an sc-identifier (but not an sc.s-modifier) that begins the first (attachable) *sc.n-sentence*. Joining in is done as follows:
- The initial *sc-identifier* of the attached sentence is omitted;
- The remainder of the *sc.n-sentence*, starting from the *sc.s-connector*, is written under the same *sc.s-identifier* but forming part of the *sc.n-sentence* to which this *sc.n-sentence* is attached. All indents in the attached *sc.n-sentence* are shifted accordingly.
  An arbitrary number of any number of branches can be formed in this way.

In essence, the semantics of the *sc.n-sentence* is the set of routes in *sc-text*, possibly intersecting and originating from a given *sc-element*.

IX. Example of the text represented in the SCg-code, SCs-code, and SCn-code

Let us consider the fragment of the *sc.g-text* shown in Figure 5. This fragment represents a class of material objects including: Earth, Moon, Sun, Mars. The material object “Moon” has two main identifiers, in Russian and English. “Earth” and “Mars” are related to “Sun” by a “revolve around*” relation. “Moon” is related to “Earth” using the “satellite*” relation.

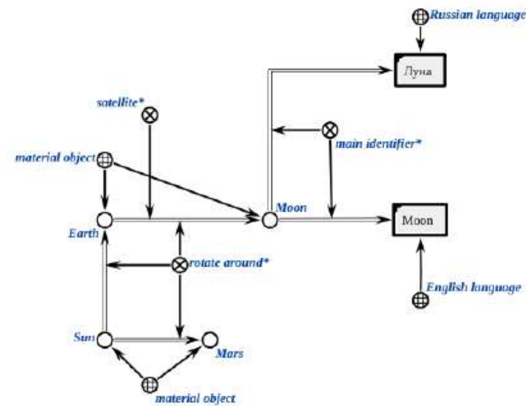

Figure 5. A fragment of the sc.g-text

```
material object -> Moon; Sun; Mars; Earth;;
Sun => rotate around*: Earth; Mars;;
Earth => rotate around*: Moon;;
Earth => satellite*: Moon;;
Moon
=> main identifier:
    [Луна]
    (* <- Russian language;; *);
    [Moon]
    (* <- English language;; *);;
```

Figure 6. A fragment of the sc.s-text

Any *sc.g-text* can easily be represented by the *sc.s-text*. Accordingly, the fragment of *sc.g-text* described above is represented in *sc.s-text* in Figure 6:

In Figure 7, a fragment of the above text in the *SCn code* is shown.

X. Conclusion

In this article, the concepts of internal and external languages of a next-generation intelligent computer system, the family of external languages of *ostis-systems* are considered. The syntax and denotational semantics of the *SCg-code*, *SCs-code*, *SCn-code* are clarified.

Examples of information constructions described with the *SCg-code*, *SCs-code*, *SCn-code* are given.

The results obtained will improve the future development of next-generation intelligent computer systems, as well as the compatibility and interoperability of the components of such systems.

Figure 7. A fragment of the sc.n-text

References

[1] V. Martynov, *Universal Semantic Code (Grammar. Dictionary. Texts)*. Minsk: Nauka i tekhnika [Science and technics], 1977.

[2] V. V. Golenkov, N. A. Gulyakina, D. V. Shunkevich, *Open technology for ontological design, production and operation of semantically compatible hybrid intelligent computer systems*, G. V.V., Ed. Minsk: Bestprint, 2021.

[3] P. Warren, P. Mulholland, T. Collins, and E. Motta, "Improving comprehension of knowledge representation languages: a case study with description logics," *International Journal of Human-Computer Studies*, vol. 122, 09 2018.

[4] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, p. 9–36, mar 1976. [Online]. Available: https://doi.org/10.1145/320434.320440

[5] J. Sowa, *Conceptual Structures: Information Processing in Mind and Machine The Systems Programming Series*, 01 1984.

[6] J. E. Rumbaugh, M. R. Blaha, W. J. Premerlani, F. Eddy, and W. E. Lorenson, "Object-oriented modelling and design," 1991.

[7] G. Schreiber, B. J. Wielinga, and J. Breuker, "Kads : a principled approach to knowledge-based system development," 1993.

[8] G. Booch, J. Rumbaugh, and I. Jacobson, "Unified modeling language user guide, the (2nd edition) (addison-wesley object technology series)," *J. Database Manag.*, vol. 10, 01 1999.

[9] D. Ausubel, "Educational psychology: A cognitive view," 01 1968.

[10] D. F. Dansereau, "The development of a learning strategies curriculum," 1978.

[11] D. H. Jonassen, K. L. Beissner, and M. Yacci, "Structural knowledge: Techniques for representing, conveying, and acquiring structural knowledge," 1993.

[12] D. F. Dansereau and C. D. Holley, "Development and evaluation of a text mapping strategy," *Advances in psychology*, vol. 8, pp. 536–554, 1982.

[13] J. M. Scandura, "Structural learning theory: Current status and new perspectives," *Instructional Science*, vol. 29, no. 4, pp. 311–336, Jul 2001. [Online]. Available: https://doi.org/10.1023/A:1011995825726

[14] G. Paquette, "Building graphical knowledge representation languages-from informal to interoperable executable models," 01 2006.

[15] A. Boriskin, M. Sadouski, D. Koronchik, I. Zhukau, and A. Khusainov, "Ontology-based design of intelligent systems user interface," *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, pp. 95–106, 2017.

[16] V. V. Golenkov, N. A. Gulyakina, D. V. Shunkevich, *Open technology for ontological design, production and operation of semantically compatible hybrid intelligent computer systems*, G. V.V., Ed. Minsk: Bestprint, 2021.

Семейство внешних языков интеллектуальных компьютерных систем нового поколения, близких языку внутреннего смыслового представления знаний

Жмырко А.В.

В работе рассматриваются понятия внешних и внутренних языков интеллектуальных компьютерных систем нового поколения. Описываются внешние языки представления знаний в рамках *Технологии OSTIS*, а именно *SCg-код*, *SCs-код*, *SCn-код*. Для каждого из внешних языков уточнены и детально рассмотрены синтаксис и денотационная семантика.

Received 10.09.2022