Ontological approach to the development of a software model of a semantic computer based on the traditional computer architecture

Daniil Shunkevich, Denis Koronchik
Belarussian State University of Informatics and Radioelectronics
Minsk, Belarus
Email: shunkevich@bsuir.by, denis.koronchik@gmail.com

***Abstract*—The paper considers an ontological approach to the development of a software model of a platform for interpreting semantic models of intelligent computer systems (a software model of a semantic computer). The architecture of the specified software model and its components are considered in detail, the principles of their implementation and the advantages of the decisions made in comparison with analogues are indicated.**

A distinctive feature of the work is the demonstration of the usage of the ontological approach to the development of software products on the example of the specified software model of a semantic computer.

***Keywords*—ontological approach, OSTIS, semantic computer, crossplatform development, graph database, semantic networks**

I. Introduction

Throughout the development of the field of artificial intelligence, attempts to create specialized hardware solutions designed to interpret a certain class of models, for example, neural network [1] or logical [2] ones, have been repeatedly made. Many of these attempts were unsuccessful, while others led to the industrial production and active usage of such specialized tools [3].

One of the main reasons for failures of such attempts was the absence of an appropriate technology that would allow implementing actively new hardware solutions in the development of intelligent systems that use models interpreted using these hardware tools. That is to say, the absence of a large number of systems, for which the usage of these models would obviously be in demand, as well as the absence of a technology that would allow developing such systems using new hardware solutions within a reasonable time were the obstacle at that time. Confirming this idea, we can introduce the relation between the rapid development of neural network models in recent years and the subsequent development of specialized hardware solutions that are used widely [3].

One of the ways that allow testing, developing and in some cases implementing new models and technologies, regardless of the availability of appropriate hardware tools, is the development of software models of these hardware tools that would be functionally equivalent to these hardware tools but at the same time would be interpreted on the basis of the traditional hardware architecture (in this article we will consider the von Neumann architecture as the dominant one at present). It is obvious that the efficiency of such software models will generally be lower than for the hardware solutions, but in most cases it is sufficient to develop the appropriate technology along with the development of hardware tools and to transfer gradually already working systems from the software model to the hardware one.

These ideas were considered when creating the *OSTIS Technology*, one of the key principles of which is the focus on a fundamentally new hardware basis for the development of intelligent computer systems – ***a semantic computer*** [4]. Currently, along with the work on creating a semantic computer, an active implementation of its software model is underway, which is currently actively used in the development of intelligent computer systems for various purposes.

The work on this software model has been carried out for a long time, and some results were published earlier by the authors [5], [6]. In contrast to these papers, the key emphasis in this paper is made on the ontological approach to the development of various kinds of products on the example of a software model of a semantic computer.

The ontological approach to software development is currently being explored as part of the Ontology Driven Software Development trend [7]. The main advantages of this approach are associated with the ability to automate the processes of analysis, development and evolution of models of software systems without taking into account the peculiarities of their implementation on specific platforms, which in turn increases the flexibility of such systems and the efficiency of their evolution (maintenance).

In this paper the *OSTIS Technology* will also be used as the basis for the ontological approach, which, in turn,

illustrates such its property as reflexivity. Therefore, as part of this paper, fragments of structured texts in the SCn-code [8] (one of the *OSTIS Technology* standards) will often be used, which are simultaneously fragments of source texts of the knowledge base that are understandable both to a human and to a machine. This allows making the text more structured and formalized while maintaining its readability.

II. Problem definition

A. Architecture of ostis-systems

The *OSTIS Technology* is based on a universal method of semantic representation (encoding) of information in the memory of intelligent computer systems called *SC-code*. Texts in the *SC-code* (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-edges*, depending on the directivity). The *SC-code alphabet* consists of five main elements, on the basis of which SC-code constructs of any complexity are built, as well as more particular types of sc-elements (for example, new concepts) are introduced.

Within the framework of the technology, several universal versions of visualization of *SC-code* constructs are also proposed, such as *SCg-code* (graphic version), *SCn-code* (non-linear hypertextual version), *SCs-code* (linear string version).

Systems built on the basis of the *OSTIS Technology* are called *ostis-systems*. Each *ostis-system* consists of a complete model of this system described by means of the *SC-code* (*sc-model of a computer system*) and a *platform for interpreting sc-models*, which in general can be implemented both in software and in hardware [9]. This ensures full platform independence of *ostis-systems*.

In turn, the *sc-model of a computer system* is conventionally divided into the *sc-model of the knowledge base*, *sc-model of the problem solver* and *sc-model of the computer system interface* (both with user and with the environment and other ostis-systems) as well as the model of abstract semantic memory (*sc-memory*), in which *SC-code* constructs are stored, and, accordingly, all the listed *sc-models* (figure 1).

Due to the availability of the *SC-code Alphabet* and the possibility of a complete description of the system using the SC-code, it becomes possible to make ostis-systems completely platform-independent. Thus, the development of an ostis-system is reduced to the development of its model and is carried out independently not only of the operating system but also of the architecture of the computer, on which the system runs. The platform, in turn, can be implemented both in the software version (in fact, in the form of a virtual machine) and in the hardware version. Therefore, the most attention within this article is paid to the software version of the platform implementation.

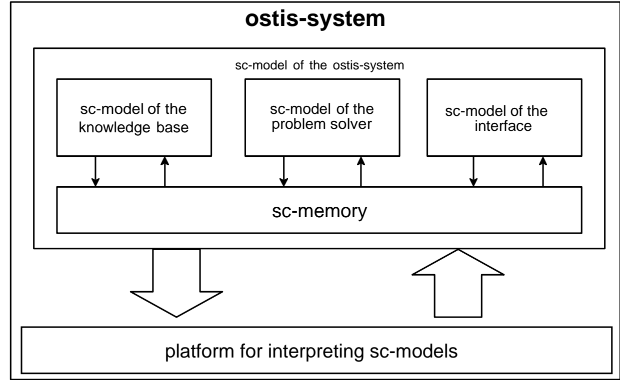

Figure 1. The architecture of the ostis-system

B. Principles which underlie the approach to the development of a software version of the implementation of the platform for interpreting sc-models

Since sc-texts are semantic networks, that is, in fact, graph constructs of a certain type, at the lower level, the problem of developing a software version of the implementation of the platform for interpreting sc-models is reduced to the development of means for storing and processing such graph constructs.

Currently, a large number of the simplest models for representing graph constructs in linear memory have been developed, such as incident matrices, adjacency lists and others [10]. However, when developing complex systems, as a rule, it is usually necessary to use more efficient models, both in terms of the amount of information required for representation and in terms of the efficiency of processing graph constructs stored in one form or another.

The most common software tools focused on storing and processing graph constructs include graph DBMS (Neo4j [11], ArangoDB [12], OrientDB [13], Grakn [14], etc.) as well as so-called rdf-storages (Virtuoso [15], Sesame [16], etc.) designed for storing constructs represented in the RDF model. To access information stored within such tools, both languages implemented within a specific tool (for example, the Cypher language in Neo4j) and languages that are standards for a large number of systems of this class (for example, SPARQL for rdf storages) can be used.

The popularity and development of such tools lead to the fact that at first glance it seems reasonable and effective to implement a *software version of the implementation of the platform for interpreting sc-models* based on one of these tools. However, there are a number of reasons why it was decided to implement a *software version of the implementation of the platform*

for interpreting sc-models from scratch. These include the following ones:

- to ensure the efficiency of storing and processing information constructs of a certain type (in this case – SC-code constructs, sc-constructs), the specificity of these constructs must be taken into consideration. Particularly, at that time, the experiments described in [5] showed a significant increase in the efficiency of the own solution compared to the existing ones;
- in contrast to classical graph constructs, where an arc or an edge can be incident only to the graph vertex (this is also true for rdf-graphs), for the SC-code, it is quite typical when an sc-connector is incident to another sc-connector or even two sc-connectors. In this regard, the existing means of storing graph constructs do not allow storing sc-constructs (sc-graphs) explicitly. A possible solution to this problem is the transition from the sc-graph to the incident orgraph, an example of which is described in [17], however, this option leads to the multiplication of stored elements by several times and significantly reduces the efficiency of search algorithms due to the need to do a large number of additional iterations;
- the basis of information processing within the OSTIS Technology is a multiagent approach, in which agents of processing information stored in sc-memory (sc-agents) react to events that occur in sc-memory and exchange information by specifying the actions they perform in sc-memory [18]. Therefore, one of the most important problems is to implement within the *software version of the implementation of the platform for interpreting sc-models* the possibility of subscribing to events that occur in the software model of sc-memory, which is currently not practically supported within the modern means of storing and processing graph constructs;
- the SC-code also allows describing external information constructs of any kind (images, text files, audio and video files, etc.), which are formally interpreted as the contents of *sc-elements* that are signs of *external files of the ostis-system*. Thus, a component of the *software version of the implementation of the platform for interpreting sc-models* should be the implementation of file memory, which allows storing these constructs in any generally accepted formats. The implementation of such a component within the modern means of storing and processing graph constructs is also not always possible.

Due to all reasons outlined, it was decided to implement a *software version of the implementation of the platform for interpreting sc-models* "from scratch", taking into account the features of information storing and processing within the OSTIS Technology. Further, the architecture of the platform implementation, the principles of storing sc-constructs in traditional linear memory as well as the implementation of tools for accessing and editing constructs stored in the sc-memory software model will be considered in detail.

III. General architecture of the software version of the implementation of the platform for interpreting sc-models

Let us consider the specification of the concept *software version of the implementation of the platform for interpreting sc-models of computer systems* in the SCn-code. Within this and other fragments, the symbol ":=" indicates alternative (synonymous) names of the described entity, which reveal in more detail some of its features.

software version of the implementation of the platform for interpreting sc-models of computer systems
:= [a software version of the implementation of the basic interpreter of logical-semantic models of computer systems]
:= [a version of the implementation of the basic interpreter of logical-semantic models of computer systems on traditional computers with the von Neumann architecture]
⊃ *web-oriented version of the implementation of the platform for interpreting sc-models of computer systems*
  ⊂ *multiuser version of the implementation of the platform for interpreting sc-models of computer systems*
  ∋ *Software version of the implementation of the platform for interpreting sc-models of computer systems*

Software version of the implementation of the platform for interpreting sc-models of computer systems
⇒ *decomposition of the software system**:
  { • *Software model of sc-memory*
    • *Implementation of the interpreter of sc-models of user interfaces*
  }

The current *Software version of the implementation of the platform for interpreting sc-models of computer systems* is a web-oriented one, that is, in terms of the modern architecture, each ostis-system is a website that is accessible online through a usual browser. Such version of the implementation has an obvious advantage – an access to the system is possible from anywhere in the world where there is an Internet connection, while no specialized software is required to work with the system. On the other hand, this version of the implementation provides the possibility to work with the system for several users parallely.

At the same time, the interaction of the client and server parts is organized in such a way that the web interface can be easily replaced with a desktop or mobile interface, both universal and specialized ones.

This version of the implementation is distributed under an open-source license; for storing source texts, the Github hosting and a collective ostis-dev account are used [19].

The implementation is crossplatform and can be compiled from source texts in various operating systems.

Figure 2 shows the current architecture of the platform for interpreting sc-models of computer systems.

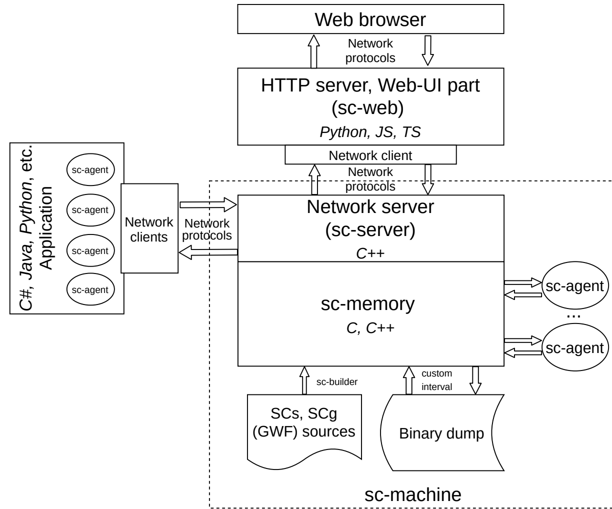

Figure 2. The architecture of the platform for interpreting sc-models of computer systems

The illustration above shows that the core of the platform is a *Software model of sc-memory* (sc-machine), which can simultaneously interact with both the *Implementation of the interpreter of sc-models of user interfaces* (sc-web [20]) and with any third-party applications using the corresponding network protocols. In terms of the general architecture, *Implementation of the interpreter of sc-models of user interfaces* acts as one of the many possible external components that interact with the *Software model of sc-memory* over the network.

Software model of sc-memory
:= [A software model of semantic memory implemented on the basis of traditional linear memory and that include storage facilities for sc-constructs and basic tools for processing these constructs, including ones for the remote access to them via appropriate network protocols]
⇐ *software model**:
sc-memory
∈ *software model of sc-memory based on linear memory*
⇒ *component of the software system**:
- *Implementation of the sc-storage and means of access to it*
- *Implementation of a basic set of platform-dependent sc-agents and their common components*
- *Implementation of the subsystem of interaction with the environment using network protocols*
- *Implementation of auxiliary tools for working with sc-memory*
- *Implementation of the scp-interpreter*

Within the current *Software model of sc-memory* [21], an *sc-storage* is understood as a component of the software model that stores sc-constructs and accesses them through a program interface. In general, the *sc-storage* can be implemented in different ways. In addition to the very *sc-storage*, the *Software model of sc-memory* also includes the *Implementation of file memory of the ostis-system* designed to store the contents of *internal files of ostis-systems*. It is worth noting that when switching from the *Software model of sc-memory* to its hardware implementation, it will be reasonable to implement file memory of the ostis-system on the basis of traditional linear memory (at least, at the first stages of the development of a *semantic computer*).

The current version of the *Software model of sc-memory* assumes the possibility of saving the memory state (snapshot) to the hard disk and further loading it from the previously saved state. This ability is necessary for restarting the system, in case of possible failures as well as when working with source texts of the knowledge base when the building from source texts is reduced to creation of a snapshot of the memory state, which is then placed in the *Software model of sc-memory*.

IV. Principles of implementation of the sc-storage

Let us consider the specification of the entity *Implementation of the sc-storage and means of access to it* in the SCn-code:

Implementation of the sc-storage and means of access to it
⇒ *component of the software system**:
- *Implementation of the sc-storage*
  ∈ *implementation of the sc-storage based on linear memory*
  ⇒ *class of software system objects**:
  segment of the sc-storage
    := [a page of the sc-storage]
    ⇒ *generalized part**:
    element of the sc-storage
- *Implementation of file memory of the ostis-system*

Within this implementation of the *sc-storage*, *sc-memory* is modeled as a set of *segments*, each of which is a fixed-sized ordered sequence of *elements of the sc-storage*, each of which corresponds to a specific sc-element. Currently, each segment consists of $2^{16} - 1 = 65535$ *elements of the sc-storage*. The allocation of *segments of the sc-storage* allows, on the one hand, simplifying an address access to *elements of the sc-storage* and, on the other hand, realizing the possibility of unloading a part of sc-memory from RAM to the file system if necessary. In the second case, the sc-storage segment becomes the minimal (atomic) unloaded part of sc-memory. The mechanism for unloading segments is implemented by the existing principles of organizing virtual memory in modern operating systems.

The maximal possible number of segments is limited by the settings of the software implementation of the sc-storage (currently, the default number is $2^{16} - 1 = 65535$ segments, but in general it may be different). Thus, technically, the maximal number of stored sc-elements in the active implementation is about 4.3×10^9 sc-elements.

By default, all segments are physically located in RAM, if there is not enough memory amount, then a mechanism for unloading part of the segments to the hard disk (the virtual memory mechanism) is provided.

Each segment consists of a set of data structures that describe specific *sc-elements* (*elements of the sc-storage*). Regardless of the type of the sc-element being described, each *element of the sc-storage* has a fixed size (currently – 48 bytes), which ensures the convenience of storing them. Thus, the maximal size of the knowledge base in the current software model of sc-memory can reach 223 GB (without taking into account the contents of *internal files of the ostis-system* stored on the external file system).

The described structure of the *Implementation of the sc-storage* is illustrated in figure 3.

Figure 4 shows an example of encoding information in the *Implementation of the sc-storage* written in the SCg-code (a graphical version of the visualization of SC-code texts). For clarity, *labels of the access level* in this example are omitted.

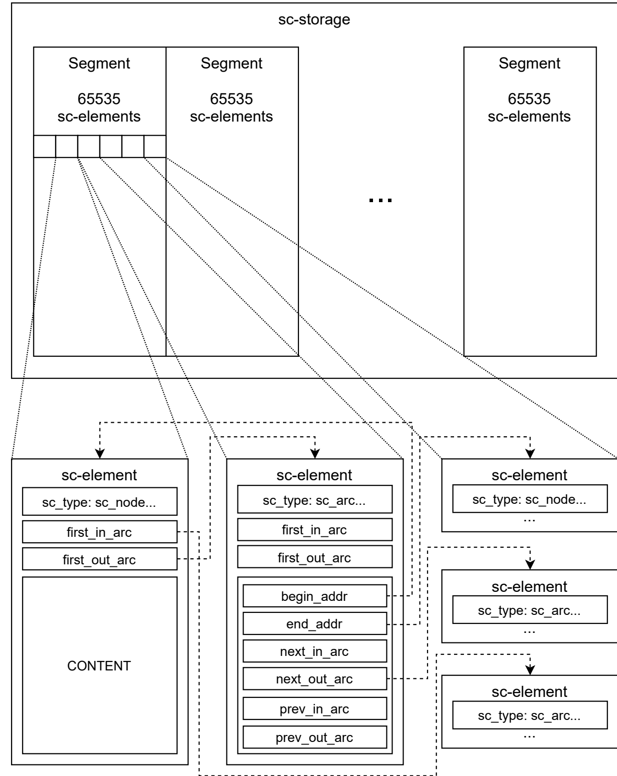

Figure 3. An example of encoding information in the Implementation of the sc-storage

sc-address

:= [an address of the element of the sc-storage that corresponds to the specified sc-element, within the current state of the Implementation of the sc-storage as part of the software model of sc-memory]

⇒ *family of relations that precisely define the structure of a given entity**:
- *segment number of the sc-storage**
- *number of the element of the sc-storage within the segment**

Each element of the sc-storage in the current implementation can be precisely specified by its address (sc-address) that consists of a segment number and a number of the *element of the sc-storage* within the segment. Thus, the *sc-address* serves as the unique coordinates of the *element of the sc-storage* within the *Implementation of the sc-storage*.

The sc-address is not taken into account in any way when processing the knowledge base at the semantic level and is only necessary to provide access to the corresponding data structure stored in linear memory at the level of the *Implementation of the sc-storage*.

In general, the sc-address of the element of the sc-storage that corresponds to the specified sc-element may change, for example, when rebuilding the knowledge base from source texts and then restarting the system. In this case, the sc-address of the element of the sc-storage that corresponds to the specified sc-element cannot change directly during the activity of the system in the current implementation.

For simplicity, we will use the term "sc-address of the sc-element", meaning the *sc-address* of the *element of the sc-storage* that uniquely corresponds to this *sc-element*.

element of the sc-storage

:= [a cell of the sc-storage]

:= [an element of the sc-storage that corresponds to the sc-element]

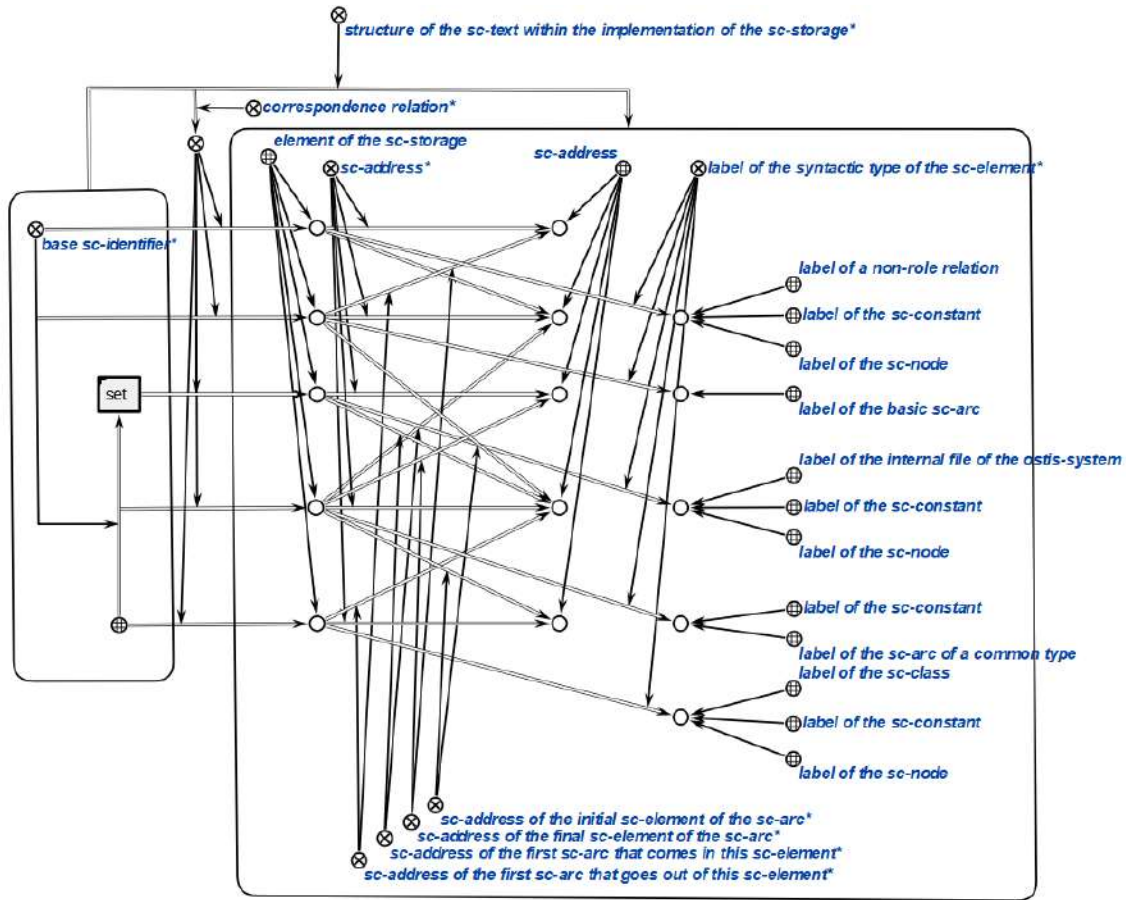

Figure 4. The Structure of the Implementation of the sc-storage

:= [an image of the sc-element within the sc-storage]
:= [a data structure, each instance of which corresponds to one sc-element within the sc-storage]
⇒ *subdividing**:
{• *element of the sc-storage that corresponds to the sc-node*
• *element of the sc-storage that corresponds to the sc-arc*
}

element of the sc-storage that corresponds to the sc-node
⇒ *family of relations that uniquely define the structure of a given entity**:
{• *label of the syntactic type of the sc-element**
• *label of the access level of the sc-element**
• *sc-address of the first sc-arc that goes out of this sc-element**
• *sc-address of the first sc-arc that comes in this sc-element**
• *contents of the element of the sc-storage**
⇒ *second domain**:
contents of the element of the sc-storage
:= [the contents of the element of the sc-storage that corresponds to the internal file of the ostis-system]
}

element of the sc-storage that corresponds to the sc-arc
⇒ *family of relations that uniquely define the structure of a given entity**:
{• *label of the syntactic type of the sc-element**
• *label of the access level of the sc-element**
• *sc-address of the first sc-arc that goes out of this sc-element**
• *sc-address of the first sc-arc that comes in this sc-element**
• *specification of the sc-arc within the sc-storage**
⇒ *second domain**:
specification of the sc-arc within the sc-storage
}

specification of the sc-arc within the sc-storage
⇒ *family of relations that uniquely define the structure of a given entity**:
{ • *sc-address of the initial sc-element of the sc-arc**
• *sc-address of the final sc-element of the sc-arc**
• *sc-address of the next sc-arc that goes out of the same sc-element**
• *sc-address of the next sc-arc that comes in the same sc-element**
• *sc-address of the previous sc-arc that goes out of the same sc-element**
• *sc-address of the previous sc-arc that comes in the same sc-element**
}

Each element of the sc-storage that corresponds to a certain sc-element is described by its syntactic type (label), and, regardless of the type, sc-addresses of the first sc-arc that comes in this sc-element and the first sc-arc that goes out of this sc-element (they can be empty if there are no such sc-arcs) are indicated.

The remainder of bytes, depending on the type of the corresponding sc-element (sc-node or sc-arc), can be used either to store the contents of the internal file of the ostis-system (it can be empty if the sc-node is not a file sign) or to store the specification of the sc-arc.

The *sc-address of the first sc-arc that goes out of this sc-element**, the *sc-address of the first sc-arc that comes in this element** and the *contents of the element of the sc-storage* may generally be missing (to be "empty", with value zero), but the size of the element in bytes will remain the same.

Each sc-node in the current implementation can have the contents (it can become an *internal file of the ostis-system*). If the size of the contents of the internal file of the ostis-system does not exceed 48 bytes (the size of the *specification of the sc-arc within the sc-storage*, for example, a small *string sc-identifier*), then this contents is explicitly stored within the element of the sc-storage as a sequence of bytes.

Otherwise, it is placed in a specially organised file memory (for its organisation a separate platform module is responsible, which in general can be arranged differently), and a unique address of the corresponding file is stored within the element of the sc-storage, which allows finding it on the file system quickly.

Currently, sc-edges are stored in the same way as sc-arcs, that is, they have a begin sc-element and an end one, the difference is only in the *label of the syntactic type of the sc-element*. This leads to some inconveniences during processing, but sc-edges are currently used quite rarely.

In terms of the software implementation, the data structure for storing the sc-node and the sc-arc remains the same, but the list of fields (components) changes in it.

In addition, as it can be seen, each element of the sc-storage (including the *element of the sc-storage that corresponds to the sc-arc*) does not store a list of sc-addresses of the sc-elements connected with it but stores sc-addresses of one outgoing and one incoming arcs, each of which, in turn, stores sc-addresses of the next and previous arcs in the list of outgoing and incoming sc-arcs for the corresponding elements.

All of the above allows:

- making the size of such a structure fixed (currently – 48 bytes) and independent of the syntactic type of the stored sc-element;
- providing the ability to work with sc-elements without taking into account their syntactic type in cases where it is necessary (for example, when implementing search requests such as “Which sc-elements are elements of this set”, “Which sc-elements are directly related to this sc-element”, etc.);
- providing the ability to access the *element of the sc-storage* in a constant time;
- providing the ability to place the *element of the sc-storage* in the processor cache, which, in turn, allows speeding up the processing of sc-constructs.

The current *Software model of sc-memory* assumes that all sc-memory is physically located on one computer. To implement a distributed version of the *Software model of sc-memory*, it is proposed to extend the *sc-address* by specifying the address of the physical device where the corresponding *element of the sc-storage* is stored.

Obviously, the type (class, kind) of the sc-element in sc-memory can be set by explicitly specifying that this sc-element belongs to the corresponding class (sc-node, sc-arc, etc.).

However, within the *platform for interpreting sc-models of computer systems*, there must be some set of *labels of the syntactic type of the sc-element* that specify the type of the element at the platform level and do not contain a corresponding sc-arc of belonging (or rather, the basic sc-arc) explicitly stored within sc-memory (its occurrence is implied, but it is not stored explicitly, since this will lead to an infinite increase in the number of sc-elements that need to be stored in sc-memory). At a minimum, there must be a label that corresponds to the *basic sc-arc* class, since an explicit indication of belonging of the sc-arc to this class generates another *basic sc-arc*.

Thus, *basic sc-arcs* that denote the belonging of sc-elements to some known limited set of classes are represented implicitly. This fact must be taken into account in a number of cases, for example, when checking whether an sc-element belongs to a certain class, when

searching for all outgoing sc-arcs from a given sc-element, etc.

If necessary, some of these implicitly stored sc-arcs can be represented explicitly, for example, in the case when such an sc-arc must be included in any set, that is, another sc-arc must be drawn into it. In this case, there is a need to synchronize the changes connected with this sc-arc (for example, with deleting it) in its explicit and implicit representation. This mechanism is not implemented in the current *Implementation of the sc-storage*.

Thus, it is impossible to completely stop using *labels of the syntactic type of the sc-element*, however, though an increase in their number raises the efficiency of the platform due to the simplifications of certain operations on the validation of types of the sc-element, but it leads to an increase in the number of situations, in which the explicit and implicit representation of sc-arcs must be taken into consideration, which, in turn, complicates the development of the platform and the development of the program code for processing the stored sc-structures.

label of the syntactic type of the sc-element
:= [a unique numerical identifier that precisely corresponds to the specified type of sc-elements and is attributed to the corresponding element of the sc-storage at the implementation level]
⇐ *second domain**:
*label of the syntactic type of the sc-element**
⊃ *label of the sc-node*
⇒ *numerical expression in the hexadecimal number system**:
[0x1]
⊃ *label of the internal file of the ostis-system*
⇒ *numerical expression in the hexadecimal number system**:
[0x2]
⊃ *label of the sc-edge of a common type*
⇒ *numerical expression in the hexadecimal number system**:
[0x4]
⊃ *label of the sc-arc of a common type*
⇒ *numerical expression in the hexadecimal number system**:
[0x8]
⊃ *label of the sc-arc of belonging*
⇒ *numerical expression in the hexadecimal number system**:
[0x10]
⊃ *label of the sc-constant*
⇒ *numerical expression in the hexadecimal number system**:
[0x20]
⊃ *label of the sc-variable*
⇒ *numerical expression in the hexadecimal number system**:
[0x40]
⊃ *label of the positive sc-arc of belonging*
⇒ *numerical expression in the hexadecimal number system**:
[0x80]
⊃ *label of the negative sc-arc of belonging*
⇒ *numerical expression in a hexadecimal number system**:
[0x100]
⊃ *label of the fuzzy sc-arc of belonging*
⇒ *numerical expression in the hexadecimal number system**:
[0x200]
⊃ *label of the permanent sc-arc*
⇒ *numerical expression in the hexadecimal number system**:
[0x400]
⊃ *label of the temporal sc-arc*
⇒ *numerical expression in the hexadecimal number system**:
[0x800]
⊃ *label of the non-binary sc-connective*
⇒ *numerical expression in the hexadecimal number system**:
[0x80]
⊃ *label of the sc-structure*
⇒ *numerical expression in the hexadecimal number system**:
[0x100]
⊃ *label of a role relation*
⇒ *numerical expression in the hexadecimal number system**:
[0x200]
⊃ *label of a non-role relation*
⇒ *numerical expression in the hexadecimal number system**:
[0x400]
⊃ *label of the sc-class*
⇒ *numerical expression in the hexadecimal number system**:
[0x800]
⊃ *label of an abstract entity*
⇒ *numerical expression in the hexadecimal number system**:
[0x1000]
⊃ *label of a material entity*
⇒ *numerical expression in the hexadecimal number system**:
[0x2000]
⊃ *label of the constant positive permanent sc-arc of belonging*
:= [a label of the basic sc-arc]

:= [a label of the sc-arc of the main type]
⇐ *intersection**:
{
- *label of the sc-arc of membership*
- *sc-constant label*
- *label of the positive sc-arc of membership*
- *permanent sc-arc label*

}
⊃ *label of the variable positive permanent sc-arc of membership*

Labels of syntactic types of sc-elements can be combined to obtain more specific label classes. In terms of software implementation, such a combination is expressed by the operation of bitwise addition of the values of the corresponding labels.

Numerical expressions of some label classes may be the same. This is done to reduce the size of the element of the sc-storage by reducing the maximal size of the label. There is no conflict in this case, since such label classes, for example, the *label of a role relation* and the *label of a fuzzy sc-arc of belonging*, cannot be combined.

It is important to note that each of the allocated label classes (except the classes obtained by combining other classes) uniquely corresponds to the ordinal number of a bit in linear memory, which can be seen by looking at the corresponding numerical expressions of the label classes. It means that the label classes are not included in each other, for example, specifying the *label of a positive sc-arc of belonging* does not automatically indicate the *label of the sc-arc of belonging*. This allows making the operations of combining and comparing labels more efficient.

Let us briefly consider the disadvantages of the current implementation of the *labels of syntactic types of sc-elements* and possible ways to eliminate them:

- At the moment, the number of *labels of the syntactic type of the sc-element* is quite large, which leads to arising a sufficiently large number of situations, in which it is necessary to take into account the explicit and implicit storage of sc-arcs of belonging to the corresponding classes. On the other hand, changing the set of labels for any purpose in the current implementation is a rather time-consuming problem (in terms of the extent of changes in the platform code and sc-agents implemented at the platform level), and an extension of the set of labels without increasing the size of the element of the sc-storage in bytes turns out to be quite impossible. The solution to this problem is to minimize the number of labels as much as possible, for example, to the number of labels that correspond to the *SC-code Alphabet*. In this case, the belonging of sc-elements to any other classes will be written explicitly, and the number of situations, in which it will be necessary to take into account the implicit storage of sc-arcs, will be minimal;
- Some labels from the current set of *labels of the syntactic type of the sc-element* are used quite rarely (for example, the *label of the sc-edge of a common type* or the *label of a negative sc-arc of belonging*), in turn, in sc-memory may exist classes that have quite a lot of elements (for example, a *binary relation* or a *number*). This fact does not allow using the efficiency of the occurrence of labels in full.
  The solution to this problem is to stop using the set of labels known in advance and switch to a dynamic set of labels (while their number can remain fixed). In this case, a set of classes expressed as labels will be formed on the basis of some criteria, for example, the number of elements of this class or the frequency of references to it.

label of the access level of the sc-element
⇐ *second domain**:
*label of the access level of the sc-element**
⇒ *generalized structure**:
{
- *label of the reading access level of the sc-element*
- *label of the writing access level of the sc-element*

}

In the current *Implementation of the sc-storage*, *labels of the access level* are used to provide the ability to restrict access of some processes in sc-memory to some sc-elements stored in sc-memory.

Each element of the sc-storage corresponds to a *label of the reading access level of the sc-element* and a *label of the writing access level of the sc-element*, each of which is expressed as a number from 0 to 255.

In turn, each process (most often one that corresponds to some sc-agent) that tries to gain access to this element of the sc-storage (read or change it) corresponds to the reading and writing access level expressed in the same numerical range. The specified access level for the process is a part of the *context of a process*. Reading or writing access to the element of the sc-storage is not allowed if the process correspondingly has a lower reading or writing access level than the element of the sc-storage that is being accessed.

Thus, value zero of the *label of the reading access level of the sc-element* and the *label of the writing access level of the sc-element* means that any process can get unlimited access to this element of the sc-storage.

V. Implementation of means for processing constructs stored in semantic memory

Let us consider the currently implemented means of processing (access and editing) constructs stored in semantic memory.

The basic means of access to constructs stored in sc-memory are *sc-iterators*.

sc-iterator
:= [ScIterator]
⇐ *class of components**:
Implementation of the sc-storage
⊃ *three-element sc-iterator*
⇒ *class of sc-constructs**:
three-element sc-construct
⊃ *five-element sc-iterator*
⇒ *class of sc-constructs**:
five-element sc-construct

From a functional point of view, *sc-iterators*, as part of the *Implementation of the sc-storage*, are a basic tool for access to constructs stored in sc-memory, which allows reading (viewing) constructs that are isomorphic to the simplest templates – *three-element sc-constructs* and *five-element sc-constructs* [8].

From the point of view of implementation, the *sc-iterator* is a data structure that corresponds to a certain additionally precised class of sc-constructs and allows using the appropriate set of functions to consistently view all sc-constructs of this class represented in the current state of sc-memory (to iterate over sc-constructs).

Each class of *sc-iterators* corresponds to some known class (template, pattern) of sc-constructs. When developing an sc-iterator, this template is precised, that is, some (at least one) elements of the template are associated with a specific *sc-element* known in advance (the starting point for the search), and other elements of the template (those that need to be found) are associated with some type of the sc-element from the types that correspond to *labels of the syntactic type of the sc-element*.

Then, by calling the corresponding function (or a class method in OOP), all sc-constructs that correspond to the received template are sequentially viewed (taking into consideration the specified types of sc-elements and sc-elements known in advance), that is, the *sc-iterator* sequentially "switches" from one construct to another as long as such constructs exist. The existence of the following construct is checked immediately before switching. In the general case, there may not be constructs that correspond to the specified template, in this case, iteration will not occur (there will be 0 iterations).

At each iteration, sc-addresses of sc-elements included in the corresponding sc-construct are written to the sc-iterator, so the found elements can be processed appropriately, depending on the problem.

Currently, a *five-element sc-iterator* is implemented on the basis of *three-element sc-iterators* and in this sense is not atomic. However, the introduction of *five-element sc-iterators* is reasonable in terms of the convenience for the developer of programs for processing sc-constructs.

sc-template
:= [ScTemplate]
⇐ *class of components**:
Implementation of the sc-storage
:= [a data structure in linear memory that describes a generalized sc-structure, which, in turn, can either be explicitly represented in sc-memory or not represented in its current state but can be represented if necessary]

Sc-iterators allow searching only for sc-constructs of the simplest configuration. To implement the search for sc-constructs of a more complex configuration as well as the generation of complex sc-constructs, *sc-templates* are used, on the basis of which the search or generation of constructs are then carried out. The *sc-template* is a data structure that corresponds to some *generalized sc-structure*, i.e., a *sc-structure* that contains *sc-variables*. Using the appropriate set of functions, the following actions can be carried out:

- search in the current state of sc-memory for <u>all</u> sc-constructs that are isomorphic to the specified template. As search parameters, the values for any of the sc-variables in the template can be specified. After the search, a set of search results will be developed, each of which is a set of pairs in the form "an sc-variable from the template – the corresponding sc-constant". This set can be empty (in the current state of sc-memory there are no constructs that are isomorphic to the given template) or contain one or more elements. Substitution of values of sc-variables can be carried out both by the sc-address and by the system sc-identifier;
- generation of an sc-construct that is isomorphic to a given template. The parameters and results of generation are developed in the same way as in the case of search, except that in the case of generation, the result is always single and a set of results is not developed.

Thus, each *sc-template* factually assumes a set of templates developed by specifying values for the sc-variables included in the initial template.

It is important to note that the *sc-template* is a data structure in linear memory that corresponds to some *generalized sc-structure* in sc-memory but is not this *generalized sc-structure* itself. It means that the sc-template can be automatically developed on the basis of the *generalized sc-structure* explicitly represented in sc-memory and also developed at the level of the program code by calling the corresponding functions (methods). In the second case, the *sc-template* will exist only in linear memory, while the corresponding *generalized sc-structure* will not be explicitly represented in sc-memory. In this case, the substitution of the values of sc-variables will be possible only by the system sc-identifier, since

the corresponding template elements will not contain sc-addresses.

When searching for sc-constructs that are isomorphic to a given template, from the point of view of efficiency, it is extremely important to consider, from which sc-element the search should be initiated. As it is known, in general, the search problem in the graph is an NP-complete problem, but the search in the sc-graph allows taking into consideration the semantics of the processed information, which, in turn, allows reducing the search time significantly.

One of the possible options for optimizing the search algorithm implemented at the moment is the ranking of three-element sc-constructs that are part of the sc-template, according to the order of search for these sc-constructs upon criterion of reducing the number of possible search options that are generated by one or another three-element sc-construct that contains sc-variables. Thus, at first, when searching, those three-element sc-constructs that initially contain two sc-constants are selected, and then those that initially contain one sc-constant. After performing the search step, the priority of sc-constructs changes, taking into account the results obtained in the previous step.

Another optimization option is based on that specific feature of formalization in the SC-code, that, in general, the number of sc-arcs that come in a certain sc-element, as a rule, is significantly lower than the number of sc-arcs that go out it. Thus, it is reasonable to initiate the search through the incoming sc-arcs.

It can be assumed that the features provided by *sc-templates* allow eliminating the usage of *sc-iterators* completely. However, this is not quite true for the following reasons:

- search and generation by a template are implemented on the basis of sc-iterators as a basic means of searching for sc-constructs within the *Implementation of the sc-storage*;
- *sc-iterators* make it possible to organize the search process more flexibly, taking into account the semantics of specific sc-elements involved in the search. For example, we can consider the fact that for some sc-elements the number of incoming sc-arcs is significantly lower than the number of outgoing ones (or vice versa). Thus, when searching for constructs that contain such sc-elements, it is more efficient to initiate a search from those sections where there are potentially fewer arcs.

context of a process within the software model of sc-memory

:= [ScContext]

:= [a context of a process run at the level of the software model of sc-memory]

:= [a meta description of a process in sc-memory run at the level of the software model of sc-memory]

:= [a data structure that contains meta information about a process run in sc-memory at the platform level]

⇐ *class of components**:
Implementation of the sc-storage

Each process that is run in sc-memory at the level of the *platform for interpreting sc-models of computer systems* (that most often corresponds to some *sc-agent* implemented at the platform level) associates with the *context of a process*, which is a data structure that describes metainformation about this process. Currently, the context of a process contains information about the reading and writing access level for this process (See the *label of the access level of the sc-element*).

When calling any functions (methods) connected with access to constructs stored in sc-memory within the process, one of the parameters is necessarily the *context of a process*.

subscription to an event in sc-memory within the software model of sc-memory

:= [ScEvent]

:= [a data structure that describes within the sc-memory software model the correspondence between the class of events in sc-memory and actions that should be performed when events of this class occur in sc-memory]

⇐ *class of components**:
Implementation of the sc-storage

To make it possible to develop sc-agents within the *platform for interpreting sc-models of computer systems*, the possibility to subscribe to an event that belongs to one of the *classes of atomic events in sc-memory** [8] is implemented, specifying the sc-element which should be associated with the event of this class (for example, the sc-element, for which an incoming or outgoing sc-arc should be shown up). A subscription to an event is a data structure that describes a class of expected events and a function in the program code that should be called when this event occurs.

All subscriptions to events are registered within the event table. With any change in sc-memory, this table is viewed and the functions that correspond to the event that occurred are run.

In the current implementation, each event is processed in a separate operating system thread, while at the implementation level a parameter is set that describes the number of maximal threads that can be run in parallel.

Thus, it is possible to implement sc-agents that respond to events in sc-memory as well as to suspend its work when running a certain process in sc-memory and wait for some event to occur (for example, create a subproblem for some group of sc-agents and wait for its solution).

To store the contents of internal files of ostis-systems, the size of which exceeds 48 bytes, files that are

explicitly stored on the file system are used, which is accessed by means of the operating system, on which the *Software version of the implementation of the platform for interpreting sc-models of computer systems* runs. The implementation of file memory is described in more detail in [5].

In addition, to implement a quick search for sc-elements by their string sc-identifiers or their fragments (substrings), an additional key-value storage is used, which corresponds to the *string sc-identifier* an *sc-address* of the *sc-element*, whose identifier is this string (in the case of the basic and system sc-identifier) or the *sc-element*, which is a sign of the *internal file of the ostis-system* (in the case of a nonbasic sc-identifier).

VI. Implementation of the basic set of platform-dependent sc-agents and their common components

Part of the *Software model of sc-memory* is the *Implementation of a basic set of platform-dependent sc-agents and their common components*, which allows the user to navigate through the knowledge base via user interface commands. This, in turn, allows beginning work with the ostis-system immediately after installing the platform and downloading the knowledge base without the need to connect any additional modules.

Implementation of a basic set of platform-dependent sc-agents and their common components
⇒ *component of the software system**:
- *Implementation of the basic set of search sc-agents*
- *Implementation of the basic mechanism for collecting junk information*
- *Implementation of the basic set of front end sc-agents*

The current implementation of the mechanism for collecting junk information contains an sc-agent that responds to the explicit addition of any sc-element to the set "junk information" and carries out the physical deleting of this sc-element from sc-memory

Implementation of the basic set of search sc-agents
⇒ *component of the software system**:
- *Implementation of an Abstract sc-agent for searching for the semantic neighborhood of a given entity*
- *Implementation of an Abstract sc-agent for searching for all entities that are particular towards a given one*
- *Implementation of an Abstract sc-agent for searching for all entities that are common towards a given one*
- *Implementation of an Abstract sc-agent for searching for all sc-identifiers that correspond to a given entity*
- *Implementation of an Abstract sc-agent for searching for basic sc-arcs that are incident to a given sc-element*

Implementation of the basic set of front end sc-agents
⇒ *component of the software system**:
- *Implementation of an Abstract sc-agent for processing user interface commands*
- *Implementation of an Abstract sc-agent for translating from an internal knowledge representation to an intermediate transport format*
  ⇒ *note**:
  [currently, an approach is used, in which, regardless of the form of external representation of information, the information stored in sc-memory is firstly translated into an intermediate transport format based on JSON, which is then processed by sc-agents of the user interface, which are part of *Implementation of the interpreter of sc-models of user interfaces*]

VII. Principles of interaction of the software model of sc-memory with external resources

The interaction of the software model of sc-memory with external resources can be carried out through a specialized program interface (API), but this option is inconvenient in most cases, because:

- it is supported only for a very limited set of programming languages (C, C++, Python);
- it requires that the client application that accesses the software model of sc-memory factually forms a whole unit with it, thus eliminating the possibility of building a distributed group of ostis-systems;
- as a consequence of the previous item, the possibility of parallel work with sc-memory of several client applications is excluded.

To make it possible to access sc-memory remotely, without taking into account the programming languages, with which a specific client application is implemented, it was decided to implement the possibility of accessing sc-memory using universal protocols that do not depend on the means of implementing a particular component or system. The binary protocol SCTP [22] and the text protocol based on JSON [23] were developed as such protocols. Further, the protocols themselves will be considered in more detail as well as the means that allow interaction on the basis of these protocols.

Implementation of the subsystem of interaction with the environment using network protocols
⇒ *component of the software system**:
- *Implementation of the subsystem of interaction with the environment using the SCTP protocol*
- *Implementation of the subsystem of interaction with the environment using protocols based on the JSON format*

SCTP is a *binary protocol* that allows performing such operations as reading (search) and editing constructs stored in sc-memory as well as track events that occur in sc-memory.

The interaction between the client and the server on the SCTP protocol is carried out by exchanging *sctp-commands*, each of which is a set of bytes intended for machine processing (but not for human perception).

SCTP
:= [The Semantic Code Transfer Protocol]
⇔ *analogy**:
HTTP
⇒ *generalized implementation**:
- *sctp-server*
- *sctp-client*

should be distinguished*
∋ {• *SCTP*
:= [The Semantic Code Transfer Protocol]
• *Stream Control Transmission Protocol*
:= [The Stream Control Transmission Protocol]
⇒ *note**:
[A transport layer protocol in computer networks developed in 2000.]
}

The stp-server processes stp-commands that come from different sctp-clients and provides their interpretation in sc-memory.

In general, stp-clients can be implemented in different programming languages and have a different programming interface. In fact, the problem of the stp-client is to convert high-level commands presented in a convenient for the programmer form into one or more low-level sctp-commands, send them to the server, wait for the sctp-result and interpret it.

sctp-command
⇒ *generalized decomposition**:
{• *sctp-command header*
:= [a part of the sctp-command that specifies its type and some additional information about it]
• *arguments of the sctp-command*
:= [a part of the sctp-command that contains its arguments and the dimension of which may vary, depending on the type of the command.]
}
⇒ *inclusion**: *example**:
- *sctp-command for deleting an sc-element with the specified sc-address*
- *sctp-command for creating a new sc-node of the specified type*
- *sctp-command for getting the initial and final elements of the sc-arc*

⇒ *note**:
[Running of each sctp-command assumes the occurrence of an sctp-result that uniquely corresponds to this command.]

The SCTP protocol has a number of advantages:
- The SCTP protocol is crossplatform;
- The SCTP protocol can be implemented quite simply in almost any programming language.

However, the SCTP protocol can be considered outdated at the moment, since it has a number of significant disadvantages:
- SCTP protocol commands are low-level (focused on work with single sc-elements or the simplest sc-constructs of 3 or 5 elements). This leads to the fact that performing even a simple transformation in the knowledge base or a content-addressable retrieval through a set of interrelated constructs is expressed in the form of a fairly large set of sctp-commands. Because for each command there is an sctp-result that is also sent over the network, this unnecessarily loads the network and greatly decreases the efficiency of the system as a whole. In addition, the efficiency of the system begins to depend heavily on the network bandwidth;
- The SCTP protocol is not designed for human perception.

Implementation of the subsystem of interaction with the environment using the SCTP protocol
⇒ *component of the software system**:
- *Implementation of the sctp-server*
- *Implementation of the sctp-client*

The *Implementation of the subsystem of interaction with the environment using the SCTP protocol* includes the *Implementation of the sctp-client* in C++, at the same time, there are other implementations of *sctp-clients* within the same software implementation of the platform, for example, within the *Implementation of the interpreter of sc-models of user interfaces*.

Due to the large number of disadvantages of the SCTP protocol, it was decided to develop another protocol

based on some universally accepted text transport format. The JSON format was chosen as such one. Currently, the existing components of the platform and specific ostis-systems are being transferred from using the SCTP protocol to the text protocol based on the JSON format. This protocol does not have its name yet.

Within the *Protocol for interaction with sc-memory based on JSON*, each command is a json-object, in which the command identifier, the type of the command and its arguments are specified. In turn, the response to the command is also a json-object, in which the command ID, its status (run successfully/unsuccessfully) and the results are specified. The structure of arguments and results of a command is determined by the type of command.

The protocol based on the JSON format has a number of advantages:

- JSON is a universally accepted open format, for working with which there are a large number of libraries for popular programming languages. This, in turn, simplifies the implementation of the client and server for the protocol built on the basis of JSON;
- The implementation of the JSON-based protocol does not apply fundamental restrictions on the dimension (length) of each command, unlike the binary protocol does. Thus, it becomes possible to use non-atomic commands that allow, for example, creating several sc-elements in one act of transferring such a command over the network. Important examples of such commands are the *Random template generation command* and the *Random template search command*;
- It can be said that the JSON-based protocol is the next step towards creating a powerful and universal request language that is similar to the SQL language for relational databases and that is designed to work with sc-memory. The next step will be the implementation of such a protocol based on one of the standards for the external display of sc-constructs, for example, *SCs-code*, which, in turn, will allow transferring entire programs for processing sc-constructs as commands, for example, in the SCP language.

VIII. Implementation of accessory tools for working with sc-memory

The *Implementation of accessory tools for working with sc-memory* is currently represented only by the *Implementation of the collector of the knowledge base from source texts written in the SCs-code*.

The collector of the knowledge base from source texts allows building a knowledge base from a set of source texts written in a limited SCs-code into a binary format perceived by the *Software model of sc-memory*. In this case, the building is possible both "from scratch" (with the destruction of a previously created memory snapshot) and an additive building, when the information contained in a given set of files is added to an already existing memory state snapshot.

In the current implementation, the collector performs "pasting together" ("merging") of sc-elements that have the same *system sc-identifiers* at the source level.

IX. Implementation of the interpreter of sc-models of user interfaces

Along with the implementation of the *Software model of sc-memory*, an important part of the *Software version of the implementation of the platform for interpreting sc-models of computer systems* is the *Implementation of the interpreter of sc-models of user interfaces*, which provides basic means for viewing and editing the knowledge base by the user, means for navigating through the knowledge base (asking questions to the knowledge base) and can be supplemented with new components, depending on the problems solved by each specific ostis-system.

Implementation of the interpreter of sc-models of user interfaces
:= [sc-web]
⇒ *programming language used**:
- *JavaScript*
- *TypeScript*
- *Python*

Figure 5 shows the Architecture of the *Implementation of the interpreter of sc-models of user interfaces*. This artwork shows the planned version of the architecture of the *Implementation of the interpreter of sc-models of user interfaces*, an important principle of which is the simplicity and uniformity of connecting with any components of the user interface (editors, visualizers, switches, menu commands, etc.). For this purpose, the Sandbox software middleware is implemented, within which low-level operations of interaction with the server part are implemented and which provides a more convenient programming interface for developers of components.

The current *Implementation of the interpreter of sc-models of user interfaces* has a number of disadvantages:

- The lack of a single unified mechanism for client-server interaction. Some components (a visualizer of sc-texts in the SCn-code, menu commands, etc.) run over the HTTP protocol, some – over the SCTP protocol using the WebSocket technology, which leads to significant difficulties in the development of the platform;
- The HTTP protocol assumes a clear separation of the active client and the passive server that responds to client requests. Thus, in practice, the server (in this case, sc-memory) cannot send a message to the client

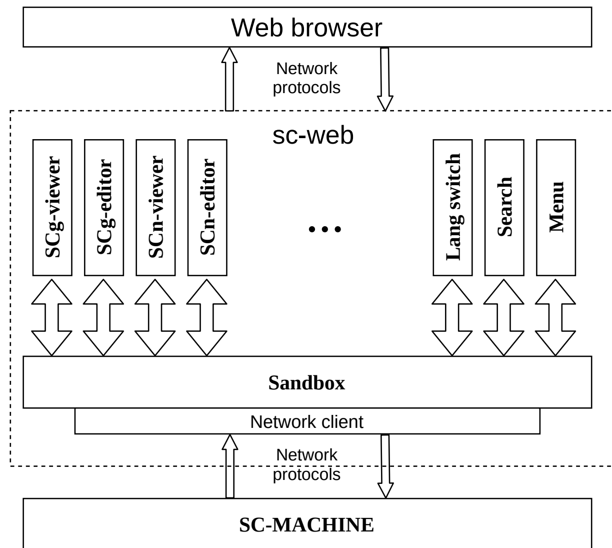

Figure 5. Architecture of the Implementation of the interpreter of sc-models of user interfaces

in for convenience, which increases the security of the system but significantly reduces its interactivity. In addition, this variant of the implementation makes it difficult to implement the multiagent approach adopted in the OSTIS Technology, in particular, it makes it difficult to implement sc-agents on the front end. These problems can be solved by constant monitoring of certain events on the front end, however, this option is ineffective;
In addition, a part of the interface factually works directly with sc-memory using the WebSocket technology, and the other one – through a middleware based on the tornado library for the Python programming language, which leads to additional dependencies on third-party libraries;

- Some of the components (for example, the search box by ID) are implemented by third-party means and are not connected with sc-memory in practice. It makes it difficult to develop the platform;
- The current *Implementation of the interpreter of sc-models of user interfaces* is focused only on conducting a dialogue with the user (in the "user question – system answer" style). Obviously necessary situations are not supported, such as running a command that does not assume an answer; an error or the absence of an answer; the need of a system to ask a question to the user, etc;
- The ability for the user to interact with the system without using special control components is limited. For example, you can ask a question to the system by drawing it in the SCg-code, but the user will not see the answer, although it will be created in memory by the corresponding agent;
- Most of the technologies used in the implementation of the platform are now outdated, which makes it difficult to develop the platform;
- The idea of platform independence of the user interface (building an sc-model of the user interface) is not implemented to the full. Currently, it is likely to be difficult to describe the sc-model of the user interface (including the exact placement, dimensions, design of components, their behavior, etc.) completely due to performance limitations, however, it is quite possible to implement the ability to ask questions to all interface components, change their placement, etc., however, these capabilities cannot be implemented in the current version of the platform implementation;
- The interface part works slowly due to the disadvantages of the SCTP protocol and some disadvantages of the implementation of the server part in Python;
- The inheritance mechanism is not implemented when adding new external languages. For example, adding a new language, even very close to the SCg-code, requires physically copying the component code and making appropriate changes, thus two components that are not connected in any way are obtained, which begin to develop independently;
- A low level of documentation of the current *Implementation of the interpreter of sc-models of user interfaces*.

Based on this, the requirements for the future (new) version of the *Implementation of the interpreter of sc-models of user interfaces*, which is currently being developed, were formulated:

- Unify the principles of interaction of all interface components with the *Software model of sc-memory*, regardless of what type the component belongs to. For example, the list of menu commands should be formed through the same mechanism as the response to the user request, and the editing command generated by the user, and the command for adding a new fragment to the knowledge base, etc;
- Unify the principles of user interaction with the system, regardless of the method of interaction and the external language. For example, it should be possible to ask questions and run other commands directly through the SCg/SCn interface. At the same time, it is necessary to take into account the principles of editing the knowledge base, so that the user cannot add new information into the compliant part of the knowledge base under the guise of asking a question;
- Unify the principles of processing events that occur when the user interacts with interface components – the behavior of buttons and other interactive components should not be set statically by third-party means but implemented as an agent, which, nevertheless, can be implemented randomly (not

necessarily at a platform-independent level). Any action performed by the user at the logical level should be interpreted and processed as the initiation of the agent;
- Provide the ability to run commands (in particular, ask questions) with a random number of arguments, including those without arguments;
- Provide the ability to display the answer to the question partially if the answer is very large and it takes a long time for it to be displayed;
- Each displayed interface component should be interpreted as an image of some sc-node described in the knowledge base. Thus, the user should be able to ask random questions to any components of the interface;
- Simplify as much as possible and document the mechanism for adding new components;
- Provide the ability to add new components based on existing ones without creating independent copies. For example, it should be possible to create a component for a language that extends the SCg language with new primitives, redefine the principles of placement of sc-texts, etc;
- Minimize dependence on third-party libraries;
- Minimize the usage of the HTTP protocol (a bootstrap loading of the general interface structure), ensure the possibility of equal two-way interaction between the server and client parts;
- Stop using the SCTP protocol completely, switch to the JSON-based one, document it.

It is obvious that the implementation of most of the above requirements is not only connected with the very version of the implementation of the platform but also requires the development of the theory of logical-semantic models of user interfaces and the clarification of the general principles of the organization of user interfaces of ostis-systems within it. However, the possibility of implementing such models in principle should be taken into account within the platform implementation.

Next, let us consider the current set of components that are included by default in the *Implementation of the interpreter of sc-models of user interfaces*. As mentioned earlier, this set can be extended by other components, depending on the problems solved by a specific ostis-system.

Implementation of the interpreter of sc-models of user interfaces

⇒ *component of the software system**:
- *User interface command menu panel*
- *Component for switching the language of identification of displayed sc-elements*
- *Component for switching the external language of knowledge visualization*
- *Search box of sc-elements by ID*
- *Panel for displaying the user dialog with the ostis-system*
- *Panel for visualization and editing knowledge*
  ⇒ *component of the software system**:
    - *Visualizer of sc.n-texts*
    - *Visualizer and editor of sc.g-texts*

User interface command menu panel contains displays of command classes (both atomic and non-atomic) that are currently available in the knowledge base and are included in the decomposition of the *Main menu of the user interface* (meaning a complete decomposition, which in general can include several levels of non-atomic command classes).

Interaction with the display of a non-atomic command class initiates a command for displaying classes of commands included in the decomposition of this non-atomic command class.

Interaction with the display of an atomic command class initiates the generation of a command of this class with previously selected arguments based on the corresponding *generalized definition of the command class* (command class template).

The *Component for switching the language of identification of displayed sc-elements* is a display of the set of natural languages available in the system. The interaction of the user with this component switches the user interface to the mode of communication with a particular user using the *base sc-identifiers* that belong to this *natural language*. It means that when displaying sc-identifiers of sc-elements in any language, for example, in the SCg- or SCn-code, the *base sc-identifiers* that belong to this *natural language* will be used. It is subject both to sc-elements displayed within the *Panel for visualization and editing knowledge* and to any other sc-elements, for example, command classes and even the *natural languages* themselves displayed within the *Component for switching the language of identification of displayed sc-elements*.

The *Component for switching the external language of knowledge visualization* is used to switch the language of knowledge visualization in the active window displayed on the *Panel for visualization and editing knowledge*. In the current implementation, SCg- and SCn-codes are supported as such languages by default as well as any other languages included in the set of *external languages of the SC-code visualization*.

The *Search box of sc-elements by ID* allows searching for sc-identifiers that contain a substring input in this box (capitalization is respected). As a result of the search, a list of sc-identifiers that contain the specified substring is displayed, when interacting with which the question “What is it?” is automatically put, the argument of which is either the sc-element itself, which has this sc-identifier (in case the specified sc-identifier is the base or system one, and thus the specified sc-element can be uniquely

defined) or the internal ostis-system file itself, which is the sc-identifier (in case this sc-identifier is not the base one).

The *Panel for displaying the user dialog with the ostis-system* displays a time-ordered list of sc-elements, which are signs of actions that are initiated by the user within the dialog with the ostis-system by interacting with displays of the corresponding command classes (that is, if the action was initiated in another way, for example, by an explicit initiation through the creation of an arc of belonging to a set of *initiated actions* in the sc.g-editor, then it will not be displayed on this panel). When the user interacts with each of the displayed signs of actions on the *Panel for visualization and editing knowledge*, a window that contains the result of performing this *action* in the language for visualization is displayed, in which it was displayed when the user viewed it the last (previous) time. Thus, in the current implementation, this panel can work only if the action initiated by the user assumes the result of this action explicitly represented in the memory. In turn, it follows that at present this panel, as the *Implementation of the interpreter of sc-models of user interfaces* in common, allows working with the system only in the "question-answer" dialog mode.

The *Panel for visualization and editing knowledge* displays windows that contain an sc-text represented in some language from a set of *external languages of the SC-code visualization* and that, as a rule, is the result of some action initiated by the user. If the corresponding visualizer supports the possibility of editing texts of the corresponding natural language, then it is at the same time also an editor.

If necessary, the user interface of each specific ostis-system can be supplemented with visualizers and editors of various external languages, which in the current version of the *Implementation of the interpreter of sc-models of user interfaces* will also be placed on the *Panel for visualization and editing knowledge*.

X. Implementation of the interpreter of programs of the base programming language of the OSTIS Technology

As a base programming language within the OSTIS Technology, including for the implementation of programs of sc-agents, an *SCP Language* [8] is proposed. The most important feature of the SCP Language is the fact that its programs are written in the same way as the knowledge they process, that is, in the SC-code. This, on the one hand, allows making ostis-systems platform-independent (clearly separate the *sc-model of a computer system* and the platform for interpreting such models) and, on the other hand, requires the occurrence of the *Implementation of the scp-interpreter* within the platform, that is, an interpreter of SCP-programs.

The structure of the *Implementation of the scp-interpreter* corresponds to the structure of an *Abstract scp-machine* (an abstract model of the scp-interpreter) considered in a number of papers, for example, in [18]. Next, let us consider this structure in the SCn-code.

Implementation of the scp-interpreter

⇐ *software implementation**:
Abstract scp-machine

⇒ *component of the software system**:
- *Implementation of an Abstract sc-agent for creating scp-processes*
- *Implementation of an Abstract sc-agent for interpreting scp-operators*
- *Implementation of an Abstract sc-agent for synchronizing the process of interpreting scp-programs*
- *Implementation of an Abstract sc-agent for killing scp-processes*
- *Implementation of an Abstract sc-agent for synchronizing events in sc-memory and its implementation*

Implementation of an Abstract sc-agent for interpreting scp-operators

⇒ *component of the software system**:
- *Implementation of an Abstract sc-agent for interpreting scp-operators for generating constructs*
- *Implementation of an Abstract sc-agent for interpreting scp-operators of the content-addressable retrieval of constructs*
- *Implementation of an Abstract sc-agent for interpreting scp-operators for deleting constructs*
- *Implementation of an Abstract sc-agent for interpreting scp-operators for checking conditions*
- *Implementation of an Abstract sc-agent for interpreting scp-operators for managing operand values*
- *Implementation of an Abstract sc-agent for interpreting scp-operators for managing scp-processes*
- *Implementation of an Abstract sc-agent for interpreting scp-operators for managing events*
- *Implementation of an Abstract sc-agent for interpreting scp-operators for processing the contents of numerical files*
- *Implementation of an Abstract sc-agent for interpreting scp-operators for processing the contents of string files*

The current *Implementation of the scp-interpreter* does not include specialized tools for working with locks, since the mechanism for locking sc-memory elements is implemented at a lower level within the *Implementation*

of the sc-storage and the mechanism for accessing it.

XI. Conclusion

The paper considers the current version of the implementation of the software implementation of the platform for interpreting sc-models of computer systems built using the OSTIS Technology. The components of this implementation, their advantages and disadvantages are considered in detail, the problems for the development of components and the platform as a whole are formulated.

It is important to note that an ontological approach was used to describe the implementation of the platform for interpreting sc-models of computer systems, which makes it possible to make such a description understandable not only to a human but also to an intelligent computer system and, ultimately, will allow changing-over to the ontological design of the platform implementation, first in software and later in hardware variants.

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.

The work was carried out with the partial financial support of the BRFFR (agreement No. F21RM-139).

References

[1] L. G. Komarcova and A. V. Maksimov, *Nejrokomp'yutery: Ucheb. posobie dlya vuzov. - 2-e izd., pererab. i dop. [Neurocomputers: A Textbook for Universities. - 2nd ed., revised and enlarged]*, ser. Informatika v tekhnicheskom universitete [Informatics at the Technical University]. Moscow: MGTU im. N.E. Baumana [Bauman Moscow State Technical University], 2004, (In Russ).

[2] (2021, Jun) PK PROLOG [PC PROLOG]. [Online]. Available: http://www.prolog-plc.ru/

[3] (2021, Jun) USB Accelerator | Coral. [Online]. Available: https://coral.ai/products/accelerator/

[4] V. Golenkov, N. Gulyakina, I. Davydenko, and D. Shunkevich, "Semanticheskie tekhnologii proektirovaniya intellektual'nyh sistem i semanticheskie associativnye komp'yutery [Semantic technologies of intelligent systems design and semantic associative computers]," *Otkrytye semanticheskie tehnologii proektirovanija intellektual'nyh sistem [Open semantic technologies for intelligent systems]*, pp. 42–50, 2019.

[5] D. N. Koronchik, "Unificirovannye semanticheskie modeli pol'zovatel'skih interfejsov intellektual'nyh sistem i tekhnologiya ih komponentnogo proektirovaniya [Unified semantic models of user interface for intelligent systems and technology for their develop]," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, V. Golenkov, Ed. BSUIR, Minsk, 2013, pp. 403–406.

[6] D. Koronchik, "Realizaciya platformy dlya web-orientirovannyh sistem, upravlyaemyh znaniyami [Implementation of web-platform for systems based on knowledges]," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, V. Golenkov, Ed. BSUIR, Minsk, 2015, pp. 89–92.

[7] H.-m. Haav, "A comparative study of approaches of ontology driven software development," *Informatica*, vol. 29, pp. 439–466, 11 2018.

[8] (2021, Jun) IMS.ostis Metasystem. [Online]. Available: https://ims.ostis.net

[9] V. Golenkov, N. Guliakina, I. Davydenko, and A. Eremeev, "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.

[10] O. P. Kuznecov, *Diskretnaya matematika dlya inzhenera: Uchebnik dlya vuzov [Discrete Mathematics for an Engineer: A Textbook for High Schools]*. Moscow: Lan', 2009.

[11] (2021, Jun) Graph Database Platform | Graph Database Management System | Neo4j. [Online]. Available: https://neo4j.com/

[12] (2021, Jun) ArangoDB, the multi-model database for graph and beyond. [Online]. Available: https://www.arangodb.com/

[13] (2021, Jun) Home | OrientDB Community Edition. [Online]. Available: https://orientdb.org/

[14] (2021, Jun) Vaticle | Home. [Online]. Available: https://vaticle.com/

[15] (2021, Jun) OpenLink Software: Virtuoso Homepage. [Online]. Available: https://virtuoso.openlinksw.com/

[16] (2021, Jun) Welcome · Eclipse RDF4J™ | The Eclipse Foundation. [Online]. Available: https://rdf4j.org/

[17] V. P. Ivashenko, N. L. Verenik, A. I. Girel', E. N. Sejtkulov, and M. M. Tatur, "Predstavlenie semanticheskih setej i algoritmy ih organizacii i semanticheskoj obrabotki na vychislitel'nyh sistemah s massovym parallelizmom [Semantic networks representation and algorithms for their organization and semantic processing on massively parallel computers]," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, V. Golenkov, Ed. BSUIR, Minsk, 2015, pp. 133–140.

[18] D. Shunkevich, "Agentno-orientirovannye reshateli zadach intellektual'nyh sistem [Agent-oriented models, method and tools of compatible problem solvers development for intelligent systems]," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, V. Golenkov, Ed. BSUIR, Minsk, 2018, pp. 119–132.

[19] (2021, Jun) OSTIS. [Online]. Available: https://github.com/ostis-dev

[20] (2021, Jun) ostis-apps/sc-web: The sc-web enchancement of https://github.com/deniskoronchik/sc-web/tree/master. [Online]. Available: https://github.com/ostis-apps/sc-web

[21] (2021, Jun) Sc-machine. [Online]. Available: http://ostis-dev.github.io/sc-machine/

[22] (2021, Jun) Sctp-protocol - sc-machine. [Online]. Available: http://ostis-dev.github.io/sc-machine/net/sctp/

[23] (2021, Jun) Websocket - sc-machine. [Online]. Available: http://ostis-dev.github.io/sc-machine/http/websocket/

Онтологический подход к разработке программной модели семантического компьютера на основе традиционной компьютерной архитектуры

Шункевич Д.В., Корончик Д.Н.

В работе рассмотрен онтологический подход к разработке программной модели платформы интерпретации семантических моделей интеллектуальных компьютерных систем (программной модели семантического компьютера). Подробно рассмотрена архитектура указанной программной модели, детально описаны ее компоненты, принципы их реализации, указаны преимущества принятых решений перед аналогами.

Отличительной особенностью работы является демонстрация применения онтологического подхода к разработке программных продуктов на примере указанной программной модели семантического компьютера.

Received 01.06.2021