A Formal Model of Shared Semantic Memory for Next-Generation Intelligent Systems

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

***Abstract*—This paper discusses in detail the formal model of semantic memory for intelligent systems, its structure, its elements, correspondences between them, rules and algorithms. The implementation based on this model is described, quantitative indicators of its efficiency are given.**

***Keywords*—shared memory, semantic memory, graph database, sc-memory, formal model of semantic memory, mathematical model of semantic memory, ostis-platform, intelligent system, unified knowledge representation, parallel information processing, semantic networks storage**

I. Introduction

Earlier in the works [1]–[3], devoted to the description of the *Software platform for intelligent systems* developed according to the principles of the *OSTIS Technology* [4] (*Software platform of ostis-systems*) — a software emulator of the future associative semantic computer [5], the software implementation of the general semantic memory (sc-memory) was considered [3], and the implementation of its programming interface was described in detail [2].

The peculiarity of previous works is that they focused not on the peculiarities of component implementation, but on approaches to describing and documenting such complex systems as the Software platform of ostis-systems [6], [7]. In this paper, the main task is to formally describe how a shared semantic memory can be realized in intelligent systems, i. e. to describe its model.

Therefore, the purpose of the current work and the novelty of this paper is to describe a formal model of shared semantic memory used in ostis-systems, allowing:

- to store information of any kind in a unified semantically compatible form;
- to efficiently process this information using a specified set of operations in both single-threaded and multi-threaded modes;
- to process this information using agents that react to events in this memory,

that is, allowing [8]:

- to represent knowledge in the form of semantically compatible knowledge bases of intelligent systems [9], [10];
- to create various types of interpreters of models of intellectual systems components, including interpreters for intellectual systems problem solvers;
- create libraries of reusable platform-dependent components to implement other components of ostis-platforms [11].

The relevance of the work is conditioned by the current state in the field of development of intelligent systems [12], namely:

- labor intensity of development of intellectual systems of various kinds;
- complexity of tasks solved in intelligent systems;
- complexity of integration of various components of intelligent systems;
- increase in the volume of processed information;
- growth of requirements to the speed of information processing;
- insufficient performance of modern open computing systems.

Further in this paper we will consider and describe the necessary and sufficient model of semantic memory for its implementation and use in solving the above problems [13]–[15].

II. Principles of implementation of ostis-platforms

All intelligent systems developed according to the principles of the *OSTIS Technology* are commonly referred to as *ostis-systems*. Each *ostis-system* consists of its own sc-model, including a knowledge base, problem solver and user interface, and an *ostis-platform* on which this sc-model is interpreted [4], [16]. Any sc-model of an ostis-system is a logical-semantic model of that system described in *SC-code*, the language of universal information encoding. By *ostis-platform* is meant a hardware-implemented computer or a software emulator of this computer for interpretation of sc-models of ostis-systems [9].

There can be a great variety of *ostis-platform* implementations on which *ostis-systems* can be interpreted, but each of such *ostis-platforms* should provide the following basic principles [1], [16], [17]:

- the development of an *ostis-system* should be reduced to the development of its sc-model (i.e. the description of the model in *SC-code* [18]) without modification of the chosen *ostis-platform* and regardless of the means by which this ostis-platform is developed;
- transfer of the sc-model of an *ostis-system* from one ostis-platform to another is limited to loading this sc-model into the memory of the ostis-platform without loss of functionality of this ostis-system;
- addition of new information should be reduced to the "gluing" of its signs with signs of existing information with further verification of the obtained information;
- processing of information in the system should be provided by changing the configuration of links between entities in this information by means of asynchronous-parallel interaction of sc-agents reacting to the occurrence of events in the shared memory.

Therefore, all ostis-systems interpreted on ostis-platforms, unlike modern computer systems, have the following features:

- unlike modern computers, where data is represented as lines of binary code, the data stored inside the memory of ostis-systems are graph constructions written in SC-code (sc-constructions), due to which [19]:
  - any kind of knowledge is written in the same form using a minimal set of syntactically indivisible elements — the minimal alphabet of a language, which can always be augmented with new syntactic elements by specifying additional syntactically distinguishable classes for the elements of that language's alphabet;
  - synonymy of entity signs is forbidden, since each entity appears in the semantic network once;
  - the meaning of information is represented by explicitly specifying the relationships between entities in that information;
  - the "gluing" of information is reduced to the "gluing" of the entity signs in that information;
  - the processing of this information does not require various means of syntactic and semantic analysis.
- information processing is based on the principles of graphodynamic and agent-oriented models, so that:
  - the process of information processing is reduced not only to changing the state of elements of the semantic network, but (!) also to changing the configuration of links between them;
  - information processing is represented and stored as a semantic network;
  - it is possible to describe and solve problems of any information complexity;
  - it is possible to realize and use any existing types and models of information processing (procedural, neural network, frame, logical, production, etc.);
  - information can be processed in parallel, i.e. different methods of problem solving can be applied simultaneously.

In other words, unlike traditional computer systems, any ostis system must be oriented towards:

- independence from the implementation of a particular ostis-platform;
- storage of information in a unified and semantically compatible form (in SC-code [18]);
- event-oriented and parallel processing of this information.

The principles of ostis-systems, first of all, should be provided by a concrete implementation of the ostis-platform. Within each *ostis-platform* there must be implemented:

- the shared semantic memory that allows [20]:
  - to store information constructions belonging to *SC-code* (sc-texts);
  - to store information constructions that do not belong to *SC-code* (images, text files, audio and video files, etc.);
  - to store subscriptions to occurrences of events in memory;
  - to initiate agents after these events appear in memory;
  - to use a programming interface to work with *SC-code* and *non-SC-code* information constructions, including:
    - operations to create, search, modify, and delete these constructions in the memory;
    - operations for subscribing to the occurrence of events in the memory;
    - operations for controlling and synchronizing processes in the memory;
    - programming interface for creating platform-dependent agents;
- the interpreter of the *SCP* asynchronous-parallel programming language, which is a platform-independent programming interface that implements platform-independent operations on the shared semantic memory.

The shared semantic memory that allows for fulfillment of all of the above mentioned tasks is called sc-memory, and the interpreter of the basic language of asynchronous-parallel programming *SCP* — *scp-interpreter*. In the context of this paper only the sc-memory model is considered. The *scp-interpreter* model and its implementation were considered in [21].

All listed principles of ostis-systems are provided in the first (basic) Software variant of the implementation of the ostis-platform — *Software platform of ostis-systems*. Drawing an analogy with modern developments in the field, the Software platform of ostis-systems can be considered as a graph database management system, for example, *Neo4J* [22]. However, unlike existing database management systems, the Software platform of ostis-systems acts as an interpreter of sc-models of semantically compatible ostis-systems. Therefore, the Software platform ostis-systems should also be considered as a software alternative for modern von Neumann computers. In general, the above mentioned features of the implemented Software platform of ostis-systems are also true for all ostis-platforms regardless of the means by which they are implemented.

An efficient implementation of sc-memory must fulfill the following requirements:

- high performance — minimizing the time spent on operations of adding, searching, modifying and deleting stored information;
- minimum memory and disk space usage for storing sc-texts;
- scalability — the ability to add computing power as the load increases without difficulty.

The following sections of this paper will discuss a possible sc-memory model and its implementation.

III. Proposed sc-memory model for next-generation intelligent systems

So, as mentioned above, in general *sc-memory* performs the following tasks:

- the task of storing sc-constructs and information constructs external to the *SC-code* (ostis-system files),
- the task of managing events and processes working on these constructs,
- the task of managing access to sc-constructions, including tasks of:
  - creating and deleting sc-elements (sc-nodes, sc-connectors, etc.);
  - searching sc-constructions by specified sc-elements;
  - getting sc-element characteristics (type, incident sc-elements);
  - adding content to sc nodes;
  - retrieving ostis-system files by known contents;
  - retrieving content from ostis-system files.

In this regard, all entities in sc-memory are:

- elements of sc-constructions:
  - sc-nodes, ostis-system files,
  - or sc-connectors (sc-arcs, sc-ribs) between sc-nodes, ostis-system files;
- elements of information constructions that are external to the SC-code:
  - string content;
  - or binary content;
- subscriptions to events in this sc-memory, that is, subscriptions to the occurrence of items in it;
- active or inactive processes in this sc-memory,
- the synchronization objects of the processes in this sc-memory,
- operations performed by processes in this sc-memory.

Thus, *the model of sc-memory* can be defined quintuple

$$SM = \langle SS,\ FS,\ RS,\ PS,\ PI \rangle,$$

where

- SS — the sc-element storage model, which is a structure of information about sc-elements,
- FS — the external information construction storage model (file memory), which is a structure of information about external information constructions,
- RS — the event subscription storage model, which is a structure of information about event subscriptions in sc-memory,
- PS — the process storage model that represents the structure of process information in sc-memory,
- PI — the set of operations over sc-memory, i.e., the programming interface of sc-memory.

A. Model of storage of sc-elements in sc-memory

The model of sc-element storage in sc-memory can be represented as

$$SS = \langle S,\ M_e, n_{s_{le}},\ n_{s_{\max}},\ n_{s_{lv}}, n_{s_{lr}},\ m_s,\ {m^n}_{s_{lv}},\ {m^n}_{s_{lr}}, SSPI \rangle,$$

where

- $S = \langle s_1,\ s_2,\ ...,\ s_i,\ ...,\ s_n \rangle,\ i = \overline{1,\ n}$ — sequence of allocated cell segments in sc-memory of fixed size n;
- $s_i = \{\langle e_{i1},\ e_{i2},\ ...,\ e_{ij},\ ...,\ e_{im} \rangle,\ n_{e_{le}},\ n_{e_{lr}},\ m_e\},\ j = \overline{1,\ m}$, — i-th segment of fixed size m, consisting of cells (elements) of sc-memory e_{ij} of fixed size k,
- $n_{e_{le}} \in (\bar{\mathbb{N}} \cup \{0\})$ — the index of the last engaged cell in the segment s_i,
- $n_{e_{lr}} \in (\bar{\mathbb{N}} \cup \{0\})$ — the index of the last released cell in the segment s_i,
- $m_e \in M$ — the object that synchronizes access to $n_{e_{le}}$ and n_{e_r};
- $M_e \subseteq E \times M$ — a dynamic oriented set of pairs of sc-elements and corresponding synchronization objects;
- $n_{s_{le}} \in (\bar{\mathbb{N}} \cup \{0\})$ — the index of the last engaged segment in sc-memory ($n_{s_{le}} = n$),
- $n_{s_{\max}} \in (\bar{\mathbb{N}} \cup \{0\})$ — the maximum number of segments in sc-memory;

- $n_{s_{lv}} \in (\bar{\mathbb{N}} \cup \{0\})$ — the index of the last vacant segment in sc-memory;
- $n_{s_{lr}} \in (\bar{\mathbb{N}} \cup \{0\})$ — the index of the last released segment in sc-memory;
- $m_s \in M$ — the object that synchronizes access to $S, n_{s_{le}}$ and $n_{s_{max}}$;
- ${m^n}_{s_{lv}} \in M$ — the object that synchronizes access to $n_{s_{lv}}$;
- ${m^n}_{s_{lr}} \in M$ — the object that synchronizes access to $n_{s_{lr}}$;
- $SSPI = \{engage,\ release\}$ — internal programming interface of sc-elements storage in sc-memory.

All allocated segments S may be free $S_f \subseteq S$ or engaged $S_e \subseteq S$. The set of free sc-memory segments S_f includes the set of vacant segments $S_v \subseteq S_f$ and the set of released segments $S_r \subseteq S_f$.

Cells in sc-memory segments E may be free $E_f \subseteq E$ and engaged $E_e \subseteq E$. The set of free sc-memory cells $E_f = \{{e^{ij}}_f |\ {e^{ij}}_f \in {s^i}_f,\ {s^i}_f \in S_f\}$ includes the set of vacant cells $E_v \subseteq E_f$ and the set of released cells $E_r \subseteq E_f$.

Consequently, the following statements hold for all sc-memory segments and cells in them:

- $E_f \cup E_e = E,\ E_f \cap E_e = \varnothing$,
- $E_v \cup E_r = E_f,\ {E}_v \cap E_r = \varnothing$,
- $S_f \cup S_e = S,\ S_f \cap S_e = \varnothing$,
- $S_v \cup S_r = S_f,\ {S}_v \cap S_r \subseteq S_f$.

For the sets of engaged and released cells, the corresponding transitions can be defined in the form of:

- operation of allocating a $engage : E_f \to E_e$, which changes the state of the cell from "released" to "engaged":

$$engage() = \begin{cases} e_{ij} \in E_e, & \text{if } \exists e_{ij} \in s_i, E_f \wedge n \leq n_{s_{\max}}, \\ \text{Error}, & \text{if } n > n_{s_{\max}} \vee |E_f| = 0; \end{cases}$$

- operation of releasing a cell in sc-memory segment $release : E_e \to E_f$, which changes the state of the cell from "engaged" to "released":

$$release(e_{ij}) = \begin{cases} e_{ij} \in E_f, & \text{if } e_{ij} \in E_e, \\ \text{Error}, & \text{if } e_{ij} \notin E_e. \end{cases}$$

The algorithm of the cell engaging operation in sc-memory segment (*engage*) can be described as follows:

- Step 1: Try to find any vacant segment s_i in the set S_v:
  - If such a segment exists, go to step 2.
  - If no such segment exists, skip to step 3.
- Step 2: Engage a new cell in the found vacant segment s_i:
  - Increase $n_{e_{le}}$ in segment s_i by 1.
  - Occupy the cell e_{ij} with index $n_{e_{le}}$ in segment s_i.
  - Return the address of the engaged cell e_{ij} and terminate.
- Step 3: Attempt to get a new segment from set S:
  - If the number of engaged segments $n_{s_{le}}$ is less than the maximum $n_{s_{max}}$, create a new segment s_i (set $n_{e_{le}}$ as 0) and add it to S.
    - Increase $n_{e_{le}}$ in segment s_i by 1.
    - Occupy the cell e_{ij} with index $n_{e_{le}}$ in segment s_i.
    - Return the address of the engaged cell e_{ij} and terminate.
  - If the maximum number of segments $n_{s_{max}}$ is reached, go to step 4.
- Step 4: Try to get the released segment s_i from the set S_r:
  - If there is no such segment, report an error and terminate.
  - If such a segment exists, proceed to step 5.
- Step 5: Engage a new cell in the found released segment s_i:
  - Get the last released cell e_{ij} by its number in segment n_{e_r}.
  - Occupy the cell e_{ij} with index n_{e_r} in segment s_i.
  - Update n_{e_r} for the next released cell.
  - Return the address of the engaged cell e_{ij} and terminate.

The algorithm of the cell releasing operation in sc-memory segment (*release*) can be described as follows:

- Step 1: Verify the correctness of the given address of cell e_{ij}:
  - If the cell address does not exist in sc-memory, terminate with an error.
  - If the cell address exists in sc-memory, proceed to step 2.
- Step 2: Using the cell address, determine the corresponding segment e_i of the cell in sc-memory and proceed to step 3.
- Step 3: Release the cell e_{ij}:
  - Update the information about the cell e_{ij}, mark it as released.
  - Update the number of the last released cell in segment n_{e_r}.
  - Go to step 4.
- Step 4: Update the information about the released segments.
  - If the released cell was the first released cell in the segment, update the information of the last released segment in sc-memory $n_{s_{lr}}$.
  - Go to step 5.
- Step 5: Terminate.

The described algorithms may include synchronization mechanisms to ensure data integrity during multi-

threaded sc-memory accesses. All synchronization operations in these algorithms can be performed using appropriate synchronization objects m_s, m_e, $m^n{}_{s_{lv}}$, $m^n{}_{s_{lr}}$. The model of the synchronization object will be discussed later.

Basically, the advantages of the described algorithms are due to the advantages of the cell engaging algorithm in sc-memory, which are as follows:

- The cell engaging algorithm in sc-memory tries to find a vacant segment before creating a new one. If there is no vacant segment, it tries to create a new one if the maximum number of segments has not been reached. This approach avoids wasting memory on creating unnecessary segments.
- By updating the number of the last released cell in a segment, the algorithm keeps track of which cells are available for reuse. This tracking ensures that the engaging process is fast and does not require searching the entire memory for a released cell.

To analyze the complexity of these algorithms, let us consider their main characteristics:

- In the algorithm of cell engaging operation in sc-memory:
  - finding a vacant segment and engaging a cell in it requires traversing many segments and checking their status, which can be accomplished in a time proportional to the number of segments;
  - creating a new segment and selecting a cell in it requires a fixed number of operations independent of the data size;
  - finding a released segment also requires traversing multiple segments.
- In the algorithm of cell releasing operation in sc-memory:
  - checking the correctness of the cell address and determining the appropriate segment can be accomplished in a time proportional to the number of segments, or faster if an efficient data structure is used to store the segments;
  - releasing a cell and updating segment information requires a fixed number of operations.

The complexity of the algorithms depends on the data structures used and how they are processed. Assuming that multiple segments are processed efficiently, for example using internal lists in released segments and released segment cells, the underlying complexity of the algorithms will be determined by the number of operations required to process each step. Thus:

- The algorithm of cell engaging operation in sc-memory can have a complexity from $O(1)$ to $O(n)$, where n is the number of segments, depending on whether a free segment is found or a new segment needs to be engaged;
- The algorithm of cell releasing operation in sc-memory basically has a complexity of $O(1)$, since most operations are performed in a fixed amount of time, except for address correctness checking, which may require $O(n)$ in the worst case.

So, an sc-element storage is a set of cells, each of which can store some sc-element (be engaged) or can be empty (free):

$$(\forall e \in E : (e \in E_e) \vee (e \in E_f)).$$

Each cell $e_{ij} \in E, e_{ij} \in s_i, s_i \in S$ has a unique internal address $a = \langle i, j \rangle \in A$. That is, the following statement always holds:

$$(\forall e_{ij} \in E, \exists! a \in A : (e_{ij} \in s_i) \wedge (s_i \in S) \wedge (a = \langle i, j \rangle)).$$

Each cell stores either an sc-node N or an sc-connector C:

$$(\forall e \in E : (e \in N) \vee (e \in EC)),$$
$$N \cup C = E, N \cap C = \varnothing.$$

It is assumed that if a cell stores some sc-element, it stores information characterizing this sc-element. Each cell in sc-memory $e \in E$ can be represented as a tuple:

$$e = \langle FI, EI, CI \rangle,$$

where

- FI — characteristics of the stored sc-element, including the type of sc-element and its states,
- EI — information about sc-elements incident with the given sc-element,
- CI — information about the number of incoming and outgoing sc-connectors for a given sc-element.

The characteristics of an sc-element FI can be represented as:

$$FI = \langle t, s \rangle,$$

where

- $t \in T$ is the syntactic type of a given sc-element (e.g., sc-node, sc-connector, ostis-system file, etc.),
- $s \in ES$ is the state of the cell (e.g., "engaged", "free", etc.).

$$T = T_n \cup T_c,$$
$$T_n = \{node,\ file\}, T_c = \{connector,\ arc,\ edge\},$$

where

- T is the set of all possible syntactic types of sc-elements;
- T_n is set of all possible syntactic types of sc-nodes, $node, file$ — actual sc-node label and ostis-system file label, respectively;

- T_c is the set of all possible syntactic types of sc-connectors, $connector, arc, edge$ — the actual sc-connector label, sc-arc label, and sc-edge label, respectively;

$$ES = \{engaged, free\},$$

where

- ES is the set of all possible cell states;
- $engaged$ — the cell is "engaged";
- $free$ — the cell is "free".

Information about incident sc-connectors EI can be represented as:

$$(\forall n \in N : (n \ni EI = \langle b_o,\ b_i \rangle)),$$

$$(\forall c \in C : (c \ni EI =$$

$$= \langle b_o,\ b_i,\ b,\ e,\ n_{bo},\ p_{bo},\ n_{bi},\ p_{bi},\ n_{eo},\ p_{eo},\ n_{ei},\ p_{ei} \rangle)),$$

where

- $b_o \in A$ is the sc-address of the initial sc-connector outgoing from the given sc element,
- $e_i \in A$ is the sc-address of the initial sc-connector incoming into the given sc-element,
- $b \in A$ is the sc-address of the initial sc-element of the sc-connector,
- $e \in A$ is the sc-address of the final sc-element of the sc-connector,
- $n_{bo} \in A$ is the sc-address of the next sc-connector outgoing from the initial sc-element,
- $p_{bo} \in A$ is the sc-address of the previous sc-connector outgoing from the initial sc-element,
- $n_{bi} \in A$ is the sc-address of the next sc-connector incoming into the initial sc-element,
- $p_{bi} \in A$ is the sc-address of the previous sc-connector incoming into the initial sc-element,
- $n_{eo} \in A$ is the sc-address of the next sc-connector outgoing from the final sc-element,
- $p_{eo} \in A$ is the sc-address of the previous sc-connector outgoing from the final sc-element,
- $n_{ei} \in A$ is the address of the next sc-connector incoming into the final sc-element,
- $p_{ei} \in A$ is the address of the previous sc-connector incoming into the final sc-element.

In this case, the following statements are true:

$$(\forall e \in E, \exists! b_o, b_i \in A : (Inc(e, b_o) \wedge Inc(e, b_i))),$$

$$(\forall\ e\ \in C,\ \exists!\ n_{bo},\ p_{bo},\ n_{bi},\ p_{bi}\ \in\ A\ :$$

$$((Inc(e,\ n_{bo}) \wedge Inc(e,\ p_{bo})) \wedge (Inc(e, n_{bi}) \wedge Inc(e,\ p_{bi})))),$$

$$(\forall\ e\ \in C,\ \exists!\ n_{eo},\ p_{eo},\ n_{ei},\ p_{ei}\ \in\ A\ :$$

$$((Inc(e,\ n_{eo}) \wedge Inc(e,\ p_{eo})) \wedge (Inc(e,\ n_{ei}) \wedge Inc(e,\ p_{ei})))),$$

where Inc is the binary relation of incidence of two sc-elements.

Information about the number of incoming and outgoing sc-connectors C can be represented as:

$$CI = \langle c_{in}, c_{out} \rangle,$$

where

- $c_{in} \in (\bar{\mathbb{N}} \cup \{0\})$ is the number of incoming sc-connectors in a given sc-element,
- $c_{out} \in (\bar{\mathbb{N}} \cup \{0\})$ is the number of outgoing sc-connectors from a given sc-element.

This information can be used to optimize the isomorphic search for sc-constructions over a given graph-template [2].

The model of storage of sc-elements in sc-memory provides:

- storage of sc-constructions, their sc-elements, characteristics and incident relations between them;
- ability to create, modify, search and delete sc-elements.

The advantages of this model are as follows:

- it provides efficient fragmentation and defragmentation of cells;
- algorithms for allocating and freeing a memory segment have asymptotic complexity from $O(1)$ to $O(n)$, where n is the number of segments that must be traversed to find a free segment.

B. Model of storage of external information constructions in sc-memory

The model of storage of external information constructions in sc-memory can be represented as

$$FS\ = \langle CH, M_s, n_{ch_{le}}, n_{ch_{max}}, m_{ch}, tr,$$
$$TSO,\ SOF,\ FSO,\ FSPI \rangle,$$

where

- $CH = \langle ch_1,\ ch_2,\ ...,\ ch_i,\ ...,\ ch_n \rangle,\ i = \overline{1,\ n}$ is the sequence of dynamically allocated file segments in sc-memory of fixed size n;
- $ch_i\ = \{\langle \langle l_{s_{i1}}, s_{i1} \rangle, ..., \langle l_{s_{ij1}}, e_{ij} \rangle, ..., \langle l_{s_{im}}, e_{im} \rangle \rangle,$ $n_{s_l},\ m_s\},\ j\ =\ \overline{1,\ m}$, — the i-th file segment of fixed size m, consisting of cells – pairs of string lengths $l_{s_{ij}}$ and strings themselves $s_{ij} \in STR$,
- $n_{s_l} \in (\bar{\mathbb{N}} \cup \{0\})$ — the index of the last engaged cell in the file segment ch_i,
- $m_s \in M$ — the object that synchronizes access to $n_{s_{le}}$;
- $M_s \subseteq CHS \times M$ — a dynamic oriented set of file and cell pairs and their corresponding synchronization objects;
- $n_{ch_{le}} \in (\bar{\mathbb{N}} \cup \{0\})$ — the index of the last engaged file segment in sc-memory ($n_{ch_{le}} =\ n$),

- $n_{ch_{max}} \in (\bar{\mathbb{N}} \cup \{0\})$ — the maximum number of file segments in sc-memory;
- $m_{ch} \in M$ — the object that synchronizes access to $CH, n_{ch_{le}} and n_{ch_{max}}$;
- $tr \subset TRM$ — rules (terms) for finding terms in strings;
- TSO — correspondence between string terms and file cell numbers with these sc-memory strings;
- SOF — correspondence between file cell numbers with strings in sc-memory and ostis-system files of which these strings are contents;
- FSO — correspondence between ostis-system files and file cell numbers with sc-memory strings, which are the contents of these strings;
- $FSPI = \{allocate,\ free,\ dump,\ load\}$ — internal programming interface of file storage in sc-memory.

Unlike sc-element storage, where the cell size is fixed and cells can be allocated in advance as some fixed sequence, this cannot be done in file storage, because cells can store strings of unknown length in advance. Therefore, the accounting of released cells, their fragmentation and defragmentation processes may be more complicated. In this connection, the problem of external fragmentation is not solved in this model, as it is solved in the sc-element storage model.

Each file cell $s_{ij} \in STR, s_{ij} \in ch_i, ch_i \in CH$ has some unique internal address $fa = \langle i, j \rangle \in FA \subset A$. That is, the following statement is always true:

$$(\forall s_{ij} \in STR, \exists! fa \in FA : (s_{ij} \in ch_i) \wedge (ch_i \in CH) \wedge$$

$$\wedge\ (fa = \langle i, j \rangle)).$$

The $allocate$ and $free$ operations can be defined for file segment cells. Their algorithms are quite simple, so they will not be considered.

File storage specifies operations to enable saving $dump$ and loading $load$ of all sc-memory.

This model is focused on the fact that any string can be partitioned into a set of terms, by which, using the TSO mapping, we can determine the indexes of strings that contain these terms [23], [24]. Then, by string indexes it is possible to obtain: from CH — the strings themselves, from SOF — the ostis-system files that contain these lines. Using the FSO mapping it is possible to find the string, which is contained by the given ostis-system file.

$$TSO = TRM \times FA,$$

$$SOF = FA \times FN, FSO = FN \times FA, FN \subset N.$$

The model of storage of external information constructions in sc-memory provides:

- storing of the contents of ostis-system files;
- setting the contents to a given ostis file;
- retrieving contents from a specified ostis file;
- retrieving ostis-files by their contents;
- obtaining ostis-system files by their content substring.

The advantages of this model are as follows:

- asymptotic complexity of adding new strings to the storage is $O(1)$ without taking into account the complexity of access time to file storage segments;
- asymptotic complexity of searching ostis-system files and their contents is $O(1)$ without taking into account the complexity of access time to segments and cells of the file storage and correspondences between ostis-system files and their contents.

C. Model of storage of subscriptions to events in sc-memory

The model of storage of subscription to events in sc-memory can be specified by the following tuple

$$RS = \langle V, m_v, RSPI \rangle,$$

where

- $V = \{v_1,\ v_2,\ ...,\ v_i,\ ...,\ v_n\}, i = \overline{1,\ n}$ is set of subscriptions to events in sc-memory of size n;
- $v_i = \langle e,\ t_v,\ a_v,\ m_v \rangle \in V$ is a subscription to an event in sc-memory;
- e is an sc-element (a cell) in sc-memory that is being "listened";
- $t_v \in T_v$ is a type of event in sc-memory;
- $ag_v \in AG$ is an agent subscribed to an event;
- $m_v \in M$ is an object that synchronizes access to subscription elements;
- $RSPI = \{subscribe, unsubscribe, notify\}$ — internal programming interface of storage of subscriptions to events in sc-memory.

All cells in sc-memory can E be listenable $E_l \subseteq E$ and non-listenable $E_{nl} \subseteq E$. For them, the following statements are true:

- $E_l \cup E_{nl} = E,\ E_n \cap E_{nl} = \varnothing$,
- $E_l \cap E_e = E_l,\ E_l \cap E_f = \varnothing$,
- $E_{nl} \cap E_e = E_{nl},\ E_l \cap E_f = \varnothing$,

$$T_v = \{aoc,\ aic,\ roc,\ ric,\ re,\ cc\},$$

where

- aoc is the event of adding an outgoing sc-connector from the listened sc-element;
- aic is the event of adding an incoming sc-connector to the listened sc-element;
- roc is the event of removing an outgoing sc-connector from the listened sc-element;
- ric is the event of removing an incoming sc-connector from the listened sc-element;
- re is the event of removing the listened sc-element;
- cc is the event of changing the content of the listened ostis-system file.

For the sets of listened and unlistened cells, the corresponding transitions can be defined in the form of:

- operation of creating a subscription to an event in sc-memory $subscribe : E \times T_v \times AG \to E_l$:

$$subscribe(e, t_v, ag_v) = \begin{cases} e_{ij} \in E_l, & \text{if } e_{ij} \in E_e \\ \text{Error}, & \text{if } e_{ij} \notin E_e; \end{cases}$$

- operation of removing a subscription to an event in sc-memory $unsubscribe : E_l \to E_{nl}$:

$$unsubscribe(e_{ij}) = \begin{cases} e_{ij} \in E_{nl}, & \text{if } e_{ij} \in E_e, \\ \text{Error}, & \text{if } e_{ij} \notin E_e. \end{cases}$$

In addition, the following operation can be defined for the set of listened events $notify : (E_l \times C) \cup E_l \to P_w$:

$$notify(e_l, c) = \begin{cases} p \in P_w & \text{if } e_l \in E_l, c \in C, Inc(e_l, c), \\ \text{Error}, & otherwise; \end{cases}$$

The notify operation can be used to notify (initiate) a process about a new event (creation of an outgoing arc c from sc-element e_l, deletion of element e_l, etc.).

The model of storage of subscriptions to events provides:

- storing of event subscriptions in sc-memory;
- ability to subscribe and unsubscribe to an event in sc-memory;
- ability to notify about an event in sc-memory.

D. Model of storage of processes in sc-memory

The model of storage of processes in sc-memory can be defined as

$$PS = \langle P_a, Q_{wp}, n_{map}, PS, PSF, PAG, PSPI \rangle,$$

where

- $P_a \subseteq P$ is the set of active processes in sc-memory;
- Q_{wp} is the queue of processes waiting to start in sc-memory;
- n_{map} is the the maximum possible number of active processes at a given time, $|Q_w p| \leq n_{map}$;
- PS is the mapping between active processes and sc-element storage segments;
- PSF is the mapping between active processes and file storage segments;
- PAG is the mapping between active processes and agents;
- $RSPI = \{activate, deactivate\}$ — internal programming interface of storage of sc-memory processes.

$$PS = (P_a \cup P_w) \times S,$$

$$PSF = (P_a \cup P_w) \times CH,$$

$$PAG = (P_a \cup P_w) \times AG.$$

The PS and PSF mappins are used to assign processes to segments of the sc-element storage and file storage. If there are enough free segments in the storage, each process is assigned separate segments in both storages.

All processes in sc-memory P can be waiting $P_w \subseteq P$, active $P_a \subseteq P$ or finished $P_f \subseteq P$.

In this case, each active and waiting process corresponds to an agent that executes it:

$$(\forall p \in P_a, \exists! ag \in AG : (\langle p, ag \rangle \in PAG)),$$

$$(\forall p \in P_w, \exists! ag \in AG : (\langle p, ag \rangle \in PAG)).$$

The following statements are true for all types of processes:

- $P_w \cup P_a \cup P_f = P, P_w \cap P_a \cap P_f = \varnothing$;
- $|P_a| \leq n_{map}$.

Transition between waiting and active processes can be defined as the function $activate : P_w \to P_a$:

$$activate(p_w) =$$

$$= \begin{cases} p_w \in P_a, & \begin{cases} ((\exists s_i \in S_f) \wedge (n_s \leq n_{s_{\max}})) \\ ((\exists ch_i \in CH) \wedge (n_{ch} \leq n_{ch_{\max}})), \end{cases} \\ \text{Error}, & \text{otherwise;} \end{cases}$$

Transition between active and finished processes can be defined as the function $deactivate : P_a \to P_f$:

$$deactivate(p_a) = \begin{cases} p_a \in P_f, & \text{if }, p_a \in P_a, \\ \text{Error}, & \text{otherwise;} \end{cases}$$

A process is considered to be finished if it is not active and not waiting:

$$((p \in P_f) \Leftrightarrow ((\neg p \in P_a) \wedge (\neg p \in P_w))).$$

The model of storage of sc-memory processes provides:

- efficient one-to-one allocation of writer-processes to sc-element storage and file storage segments;
- queuing new processes when the device's processing power is limited, and activating processes from the queue when some active process has finished its work.

E. Model of coordinated access (synchronization) of processes to sc-memory

Each synchronization object $m \in M$ can be represented as [25], [26], [27]:

$$m = \langle c_{ar}, f_{aw}, Q_{rw}, m_u \rangle,$$

where

- c_{ar} is the active reader count;
- f_{aw} is the flag that shows whether a reader is active at a given moment;
- Q_{rw} is the queue of readers and writers;
- m_u is the object used to synchronize access to elements of a given synchronization object.

The queue of readers and writers is a sequence of requests to acquire a particular resource:

$$Q_{rw} = \langle q_1, q_2, ..., q_j, ..., q_m \rangle.$$

Each request q_j includes a unique thread identifier id_j, a thread type (reader or writer) tt_j, and a condition variable that allows messages to be exchanged between processes (threads) cv_j:

$$q_j = \langle id_j, tt_j, cv_j \rangle.$$

This queue ensures that no thread is left hungry.

To coordinate access to data structures in sc-memory, mechanisms for acquiring and releasing resources for reader-threads P_r (hereinafter — readers) and writer-threads P_w (hereinafter — writers) are required.

$$P = P_r \cup P_w.$$

These mechanisms should include:

- a reader resource acquisition operation (*acquire_read*) that allows a reader-thread to acquire a synchronization object to start reading the resource and suspend the execution of all other writer-threads while there are readers in the reader-writer queue;
- a reader resource release operation (*release_read*) that allows a reader-thread to release the synchronization object after finishing reading the resource and notify all other writer-threads to execute if there are no active readers in the reader-writer queue after releasing the synchronization object;
- a writer resource acquisition operation (*acquire_write*) that allows a writer-thread to acquire a synchronization object to start modifying the resource and suspend execution of all other reader-threads and writer-threads while there is an active writer in the reader-writer queue;
- a writer resource release operation (*release_write*) that allows a writer-thread to release the synchronization object after the resource modification is complete and notify all other reader-threads and writer-threads to execute if there are no active readers in the reader-writer queue after the synchronization object is released;
- a reader multi-resource acquisition operation (*acquire_read_n*) that allows a reader-thread to acquire multiple synchronization objects for reading in the order necessary to prevent deadlocks;
- a reader multi-resource release operation (*release_read_n*) that allows a reader-thread to release multiple synchronization objects in the reverse order of acquisition;
- a writer multi-resource acquisition operation (*acquire_write_n*) that allows a writer-thread to acquire multiple synchronization objects to modify in the order necessary to prevent deadlocks;
- a writer multi-resource release operation (*release_write_n*) that allows a writer-thread to release multiple synchronization objects in the reverse order of acquisition.

Allocation of sc-memory to writers can be done segment-by-segment using a specialized table T_{ps}, which allows to determine whether a given vacant sc-memory segment has been acquired by another writer:

$$s : S \to S_v,\ T_{ps} \subseteq P_w \times S_v.$$

Let us recall that a free sc-memory segment can be either a segment with vacant cells or a segment with released cells. When allocating sc-memory, the first thing that is done is to search for vacant segments that are not used by other writers. If no such segments are found, new segments are allocated. If there is no available space in the sc-memory for new segments, writers can use segments from the list of engaged vacant segments.

To ensure coordinated read access to segments, each segment contains a unique synchronization object.

$$m_s : S \times M \to S_m,$$

$$m_{ch} : CH \times M \to CH_m.$$

In addition to segments, synchronization objects are also temporarily assigned to sc-memory cells and events to be registered in it. Synchronization objects of sc-memory cells can be stored in a specialized table T_{em}. These objects are used to synchronize access to the sc-element information contained in the sc-memory element:

$$T_{em} \subseteq A \times M.$$

These objects synchronize the subscription and unsubscription to events through a single table, as well as the initiation of the sc-agents themselves

$$v_i = \langle t_v^i,\ A_v^i,\ m_v^i \rangle, t_v^i \in T_v, A_v^i \subseteq A_v, m_v^i \in M,$$

$$v_m : V \times M \to V_m.$$

The model of synchronization of process access to sc-memory provides:

- parallel access to sc-memory, i.e. the possibility of parallel execution of actions in sc-memory without violating correctness of data structures in it;

- absence of deadlocks, races and hungry processes in sc-memory;
- fast parallel creation of sc-elements in sc-memory due to distribution of processes over sc-memory segments;
- fast non-blockable parallel search of sc-constructions provided that no other operations are performed on these sc-constructions.

F. Model of sc-memory programming interface

The model of sc-memory programming interface can be defined as follows

$$PI = N^T \times C^{E\times E\times T} \times F^{N\times L}\times$$

$$\times\{E^{E\times\{E\cup T\}\times E}\cup\ E^{\{E\cup T\}\times T\times E}\cup E^{E\times T\times\{E\cup T\}}\}\times$$

$$\times\ T^E\times\ F^L\times L^F\times\{\top,\bot\}^E\times\ V^{E\times T_v\times AG}\times\{\top,\bot\}^V$$

where

- N^T is the operation of creating an sc-node with the specified type;
- $C^{E\times E\times T}$ is the operation of creating an sc-connector between two given sc-elements with the specified type;
- $F^{N\times L}$ is the operation of setting the contents to an sc-node;
- $\{E^{E\times\{E\cup T\}\times E}\cup\ E^{\{E\cup T\}\times T\times E}\cup E^{E\times T\times\{E\cup T\}}\}$ are operations of searching for three-element sc-constructions by given first and/or second and/or third sc-elements;
- T^E is the operation of obtaining the type of the given sc-element;
- F^L is the operation of obtaining ostis-system files by their contents;
- L^F is the operation of obtaining the content from the ostis-system file;
- $\{\top,\bot\}^E$ is the operation of deleting the specified sc-element.
- $V^{E\times T_v\times AG}$ — operation of creating a subscription to an event in sc-memory;
- $\{\top,\bot\}^V$ — operation of removing a subscription to an event in sc-memory;

Let us consider some of the algorithms of the described operations. The algorithm of the operation of creating an sc-node with the specified type N^T can be described as follows:

- Step 1: Verify that the specified sc-element type is a subtype of sc-node.
  - If the specified sc-element type is not a subtype of sc-node, then terminate the algorithm with an error.
  - Otherwise, proceed to step 2.
- Step 2: Allocate a new sc-memory cell for the sc-node.
  - If the sc-memory is engaged, terminate the algorithm with an error.
  - Otherwise, proceed to step 3.
- Step 3: Set the type for the cell as sc-node with the specified type, go to step 4.
- Step 4: Return the resulting sc-address of the sc-node and terminate.

The algorithm for the operation of creating an sc-connector between two given sc-elements with the specified type $C^{E\times E\times T}$ can be described as follows:

- Step 1: Verify that the specified sc-element type is a subtype of sc-connector.
  - If the specified sc-element type is not a subtype of sc-node, then terminate the algorithm with an error.
  - Otherwise, proceed to step 2.
- Step 2: Check that the sc-addresses of the start and end sc-elements are valid.
  - If the sc-address is not valid, then terminate the algorithm with an error.
  - Otherwise, proceed to step 3.
- Step 3: Allocate a new sc-memory cell for the sc-connector.
  - If the sc-memory is engaged, terminate the algorithm with an error.
  - Otherwise, proceed to step 4.
- Step 4: Set the type for the cell as sc-connector with the specified type, go to step 5.
- Step 5: Add the sc-connector to the list of outgoing and incoming arcs of the start and end sc-elements, go to step 6.
- Step 6: Notify the outgoing and incoming arc addition events, go to step 7.
- Step 7: Return the resulting sc-connector address and terminate.

The algorithm of the operation of deleting a given sc-element $\{\top,\bot\}^E$ can be described as follows:

- Step 1: Attempt to acquire a cell in sc-memory by the sc-address of the sc-element.
  - If the cell is not found, terminate the algorithm with an error.
  - Otherwise, proceed to step 2.
- Step 2: Initialize the stack to remove sc-elements, go to step 3.
- Step 3: Put the sc-address of the sc-element to be deleted into the deletion stack, go to step 4.
- Step 4: Place the sc-addresses of all sc-connectors for which the given sc-element is the start or end sc-element and the sc-addresses of all connectors for which the found sc-connectors are the start or end sc-elements on the deletion stack, go to step 5.
- Step 5: While the deletion stack is not empty, set the cells for the sc-elements as released and release

all those cells.
  - For all sc-connectors to be deleted, notify outgoing and incoming sc-connector deletion events.
  - For all sc-cells to be deleted, notify sc-cell deletion events.
  - Go to step 6.
- Step 6: Destroy the stack for sc-element deletion, go to step 7.
- Step 7: Terminate.

The algorithm for the operation of setting the content to a given sc-node $F^{N\times L}$ can be described as follows:

- Step 1: Attempt to retrieve a cell in sc memory by the sc address of the sc node.
  - If the cell is not found, then terminate the algorithm with an error.
  - Otherwise, proceed to step 2.
- Step 2: Change the sc-node type to an ostis-system file, go to step 3.
- Step 3: Add the string to the file storage of sc-memory.
  - Add the string to a free segment of the file storage.
  - Assign matches between this string and the specified ostis file.
  - Notify the event of changing the content of the ostis-system file.
  - Go to step 4.
- Step 4: Terminate.

The principles of search operations were discussed in [2].

This programming interface provides all the necessary functionality for working with sc-constructions, file constructions, events and processes in the memory.

G. Conclusions

The proposed model of the shared semantic memory includes a formal description of the following (!):

- how to represent, store, and process graph and string constructions, events, and processes in the memory;
- how to ensure efficient execution of operations in this shared memory;
- how to coordinate multiple processes running at the same time on the same memory location,
- how to efficiently utilize the available computing power, etc.,

and allows to (!):

- efficiently organize joint storage of graph constructions and string content of external information constructions not represented as a graph, using graph-dynamic and event-driven models;
- efficiently manage the address space, i.e. distribute information about these constructions the in memory in the most effective way;
- efficiently allocate processes to work with these constructions in single-threaded and multi-threaded environments;
- provide coordinated (synchronized) execution of several processes in one memory;
- ensure consistency of operations at the level of representation and processing of data in the memory.

This model has many merits, but the following issues remain unresolved:

- how to ensure the security of information storage and processing, that is:
  - how to ensure access rights for constructions stored in the memory;
  - how to efficiently process and assign these access rights to processes;
  - and more;
- how to ensure consistency of operations at the knowledge representation and processing level, i.e.:
  - how to implement transactions for graphs;
  - how to ensure the integrity and atomicity of some group of operations on a subgraph;
  - how to ensure error-free execution of these transactions;
  - and more;
- how to ensure the storage and processing of information in teams of intelligent systems [28], that is:
  - how to organize storage and processing of information in distributed memory, i.e. in memory not on one device, but on multiple devices;
  - how to efficiently organize data transfer over a network between several devices;
  - and more;
- how to organize the configuration of memory components from the memory itself.
- and so on.

Nevertheless, the results of this paper are very significant for future work. These questions will be discussed in the following papers. Let us consider some obtained quantitative characteristics of the implementation of the proposed model.

IV. The software implementation of sc-memory for next-generation intelligent systems

A. Description of the sc-memory implementation

The current version of sc-machine is implemented on the Linux operating system (Ubuntu-22.04) [29] and is available on GitHub [30]. When developing sc-machine according to the described model, we used modern development environments (CLion, VSCode), containerization tools (Docker), programming languages (C, C++, CMake), as well as standard libraries and frameworks supplied together with compilers of the programming languages used. The development was based on the models and tools described in the previous section, as

well as the *OSTIS Technology Standard* described in the current version of the *OSTIS-2023 monographs* [9].

The current Software implementation of sc-memory has the following features:

- The memory allocation and destruction mechanisms of the GLib library are used to manage dynamic memory.
- Prefix trees [31] and linked lists are used as data structures to store *information constructions* that do not belong to *SC-code*. The reasons for that are as follows [32]:
  - prefix structures are fairly easy to understand and minimal in their syntax;
  - prefix structures are convenient enough to store and handle "key-value" relations;
  - access to a value by a key occurs in the worst case for the length of this key [33].

  The Implementation of file memory allows storing and searching any kind of information constructs (including binary files).
- To synchronize processes in sc-memory, monitors are implemented and used [34], [27]. They provide:
  - locking mechanisms to prevent multiple processes from simultaneously accessing shared resources, eliminating the possibility of mutual exceptions, race conditions, and data access conflicts;
  - high-precision time synchronization between processes that allows them to work in a coordinated mode, which eliminates the possibility of some processes being hungry.

  The implementation of monitors uses mutexes, condition variables, and queues.
- The current *Programming interface of the Software implementation of sc-memory* allows:
  - implement platform-dependent components to a necessary and sufficient extent, almost independently of sc-memory implementation.
  - implement basic tools for designing platform-independent ostis-systems.
- The current Implementation of sc-memory is fully consistent with the current Implementation of scp-interpreter.

In general, *sc-memory* can be implemented in different ways. For example, another variant of *ostis-platform sc-memory* can be realized by a program implementation of *Neo4j DBMS*. The difference between such a possible *sc-memory* implementation and the current one is that the storage of *graph constructions* and the control of the flow of actions over them should be realized more by means provided by *Neo4j DBMS*, while the representation of *graph constructions* should be implemented in its own way, because it depends on the *SC-code syntax*. [18].

B. Efficiency of sc-memory operations

The current Software implementation of sc-memory in the Software platform for ostis-systems allows to store and represent *sc-constructions*, external *information constructions* not belonging to *SC-code*, as well as to control and coordinate processes in it.

The results of sc-memory operations testing, which includes the implementation of the process control model, showed that parallel execution of sc-memory operations is efficient when the number of operations is large enough (e.g., 1,000,000 operations) (Table 1).

Table I
Efficiency of using 4 physical threads to perform 1,000,000 sc-memory operations compared to 1 physical thread

Number of physical threads	1 thread	4 threads	
	Response time, ms	Response time, ms	Speedup, times
Operations of addition (modification)			
Operation of sc-node creation	958,025	369,680	2.591
Operation of sc-connector creation	1,299.740	787.001	1.652
Operation of adding content to ostis-system file	29,885.500	9,555.450	3.128
Operations of search			
Operation of searching sc-connectors outgoing from a given sc-element	642.378	203.005	3.164
Operation of searching an ostis-system file by its contents	1,608.650	928.555	1.732
Operations of deletion			
Operation of deleting an sc-element	1,850.950	1,746.270	1,060
Operation of deleting sc-connectors outgoing from a given sc-element	1,704.620	2,115.500	0.806

Testing and evaluation of the effectiveness of the ostis-systems software platform were conducted on one of its latest versions — 0.9.0. This version of the platform solved the problem of controlling processes in the shared semantic memory. During the testing we calculated the main efficiency (performance) indicators of operations over sc-memory in single-threaded and multithreaded environments: response time and throughput, and also calculated the speedup [35] obtained by using parallelism when performing a group of operations of the same class over sc-memory [36]. The computer used was an

HP ProBook Hewlett Packard laptop with a *Intel(R) Core(TM) i7-4900MQ* processor (4 cores with 2 threads) with a configured core frequency of 3.20 GHz, 16 GB RAM, and 256 GB SSD.

At the same time, parallel execution of a small number of operations over sc-memory (for example, 100 or 10,000 operations) in some cases can be worse than their sequential execution (Table 2).

This behavior is related to the peculiarities of the control mechanisms of the processes in the shared semantic memory, the classes of operations to be performed, and their specified input values in the context of the problem to be solved. For example, all sc-construction search operations with the same sc-elements, executed in parallel, do not block each other. For example, the speed of parallel execution of operations on ostis-system files depends on the size of the buffer used when reading external information constructions and writing them to disk, as well as on the length of the information constructions themselves.

Table II
Efficiency of using 4 physical threads to perform 100 sc-memory operations compared to 1 physical thread

Number of physical threads	1 thread	4 threads	
	Response time, ms	Response time, ms	Speedup, times
Operations of addition (modification)			
Operation of sc-node creation	0.099	1.306	0.076
Operation of sc-connector creation	0.150	0.422	0.356
Operation of adding content to ostis-system file	9.521	4.128	2.307
Operations of search			
Operation of searching sc-connectors outgoing from a given sc-element	0.530	0.241	2.200
Operation of searching an ostis-system file by its contents	0.339	1.453	0.233
Operations of deletion			
Operation of deleting an sc-element	0.144	1.494	0.096
Operation of deleting sc-connectors outgoing from a given sc-element	0.182	0.938	0.194

The figure 1 shows Dependence of speedup coefficient from parallel execution of a group of operations of the same class on 4 processes on the number of operations in this group, and the figure 2 shows Dependence of the execution time of a group of operations on the number of processes used.

C. Efficiency of network operations over sc-memory

Network access to sc-memory is provided by the server subsystem of the ostis-systems software platform, implemented on the basis of Websocket and JSON languages (protocols) and providing network operations (commands) over sc-memory [3].

In the process of testing the implementation, the throughput of its commands was calculated. During the load testing a test client system implemented in C++ was used. The same device was used as the device used for testing operations over sc-memory. As a result, it was found out that when sending 1000 different commands: *commands for creating sc-elements*, *commands for processing contents of ostis-system files* and *commands for deleting sc-elements* — the time spent on their processing did not exceed 0.2 seconds. At the same time, in some cases it took no more than 0.14 seconds to process 1000 *commands for creating sc-elements*, while for *commands for deleting sc-elements* it took no more than 0.12 seconds, *commands for processing the contents of ostis-system files* — no more than 0.10 seconds, *commands to search for sc-constructions isomorphic to a given five-element graph-template* — no more than 0.45 seconds.

D. Conclusions

From the test results, it is clear that the current implementation of the ostis-systems software platform is an effective means of processing distributed information using both the software interface and the network interface and communication protocols.

The current *Implementation of sc-memory* provides:

- stability in single-threaded and multi-threaded modes;
- dast speed of work in single-threaded and multi-threaded modes;
- reliability of knowledge and data storage and processing in single- and multi-threaded modes.

The proposed shared semantic memory model enables efficient tracking and synchronization of parallel data accesses. The implementation of this model demonstrates a significant increase (by 2-3 orders of magnitude) in the throughput of parallel task execution compared to previous versions of the platform. However, to ensure (causal, sequential) consistency of processes and their operations, besides the data level, it is necessary to manage the knowledge level [37] .

V. Conclusion

In this paper, a model and implementation of the shared semantic memory has been proposed and discussed in detail, including (!):

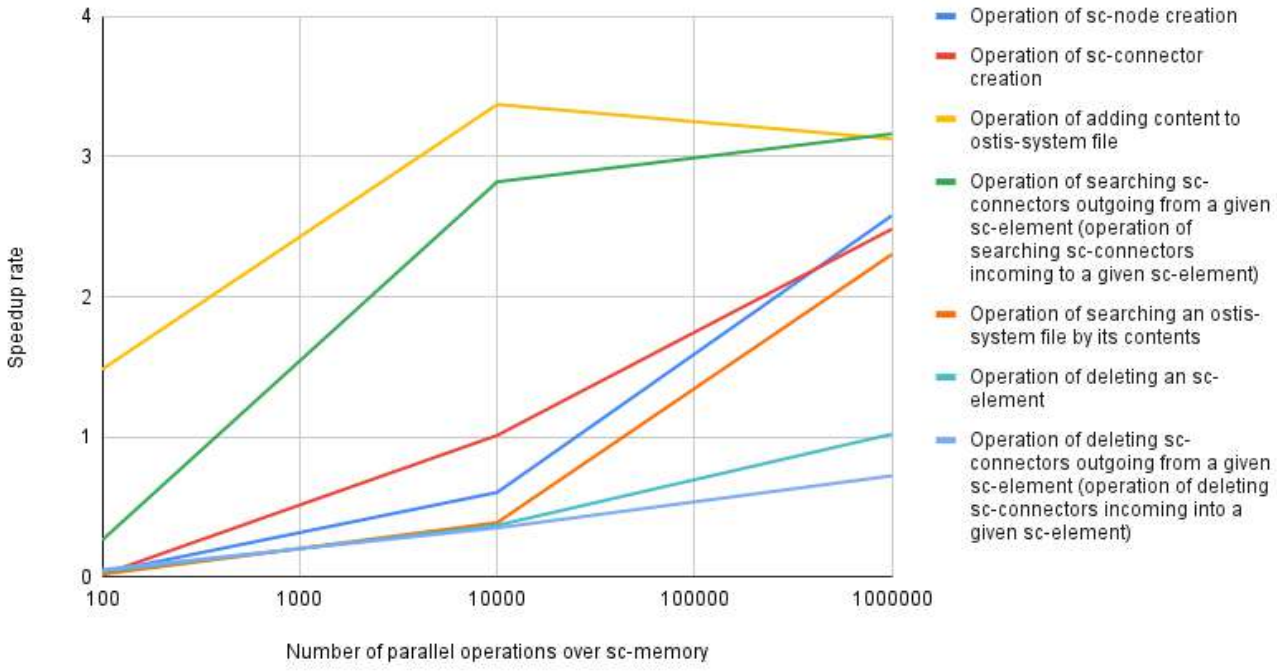

Figure 1. Dependence of speedup coefficient from parallel execution of a group of operations of the same class on 4 processes on the number of operations in this group

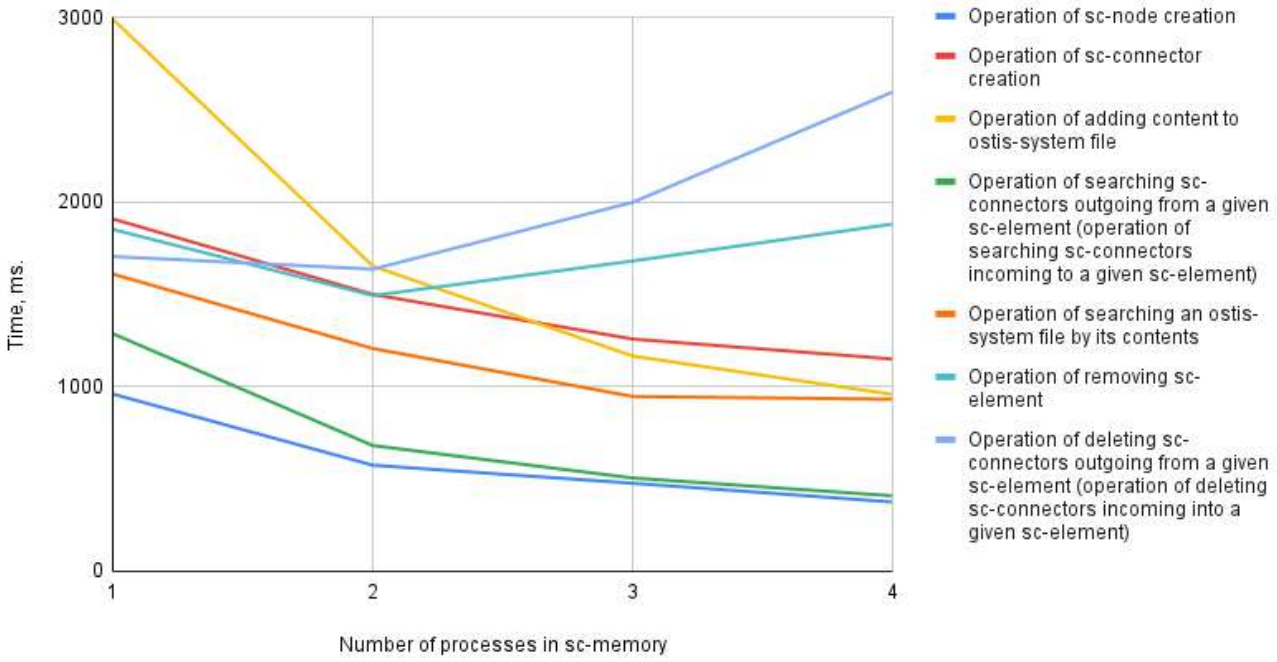

Figure 2. Dependence of the execution time of a group of operations on the number of processes used

- a storage for unified representation and processing of graph constructions;
- a storage for unified representation of string constructions used as file contents in graph constructions;
- a storage for managing events in this memory;
- a storage for managing processes running in this memory;
- a set of operations for working with this memory.

The proposed model of the shared semantic memory includes:

- models and algorithms for allocating and releasing cells in this memory, providing:
  - reusability of the released memory segments;
  - ability to utilize new vacant memory segments;
- Models and algorithms to efficiently allocate processes in this memory;
  - rapid parallel creation of elements in the memory by allocating processes over the segments of the memory;
  - fast unblockable parallel search of constructions, provided there are no other operations on these constructions.
- Models and algorithms for managing subscriptions to events in this memory;
- Models and algorithms for synchronizing the execution of processes in the shared memory sections, providing:
  - parallel access to sc-memory, i.e. possibility of parallel execution of actions in sc-memory without violating correctness of the data structures in it;
  - absence of deadlocks, races and hungry processes operating in sc-memory.

Promising directions to further this line of work are:

- development of a model for distributed unified representation and processing of information in the unified semantic memory;
- development of a model for representation and storage of platform-dependent agent programs;
- development of a consistency model to ensure correctness of agents' operation on constructions in the memory;
- development of a model of memory configuration from the memory itself.

In addition, other equally important areas of work are:

- improving the documentation of the current Implementation of sc-memory and the current Software implementation of ostis-platform;
- improvement of methodologies and tools for developing documentation of software systems;
- improvement and mass distribution of the Software implementation of ostis-platform and intelligent systems developed on its basis.

The formally described model of semantic memory is consistent with the previously described ontological model of this memory [3]. The author of this paper believes that the used approach to modeling of complex objects will help to simplify the understanding of the operation of intelligent systems developed according to the principles of the OSTIS Technology.

Acknowledgment

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

References

[1] N. Zotov, "Design principles, structure, and development prospects of the software platform of ostis-systems," *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, pp. 67—-76, 2023.

[2] ——, "Implementation of information retrieval subsystem in the software platform of ostis-systems," *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, pp. 77–94, 2023.

[3] ——, "Software platform for next-generation intelligent computer systems," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*. BSUIR, Minsk, 2022, pp. 297—-326.

[4] V. Golenkov, N. Guliakina, and D. Shunkevich, *Otkrytaja tehnologija ontologicheskogo proektirovanija, proizvodstva i jekspluatacii semanticheski sovmestimyh gibridnyh intellektual'nyh komp'juternyh sistem [Open technology of ontological design, production and operation of semantically compatible hybrid intelligent computer systems]*, V. Golenkov, Ed. Minsk: Bestprint [Bestprint], 2021.

[5] V. Golenkov *et al.*, "Associative semantic computers for intelligent computer systems of a new generation," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open Semantic Technologies for Intelligent Systems (OSTIS)]*, vol. 7. BSUIR, Minsk, 2023, pp. 39–60.

[6] V. Golenkov, "Ontology-based design of intelligent systems," *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, pp. 37–56, 2017.

[7] D. Shunkevich, "Ontology-based design of hybrid problem solvers," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*. Cham: Springer International Publishing, 2022, pp. 101–131.

[8] S. Auer, V. Kovtun, M. Prinz, A. Kasprzik, M. Stocker, and M. E. Vidal, "Towards a knowledge graph for science," in *Proceedings of the 8th international conference on web intelligence, mining and semantics*, 2018, pp. 1–6.

[9] V. Golenkov, Ed., *Tehnologija kompleksnoj podderzhki zhiznennogo cikla semanticheski sovmestimyh intellektual'nyh komp'juternyh sistem novogo pokolenija [Technology of complex life cycle support of semantically compatible intelligent computer systems of new generation]*. Bestprint, 2023.

[10] A. Palagin and N. Petrenko, "To the question of system-ontological integration of subject area knowledge," *Mathematical Machines and Systems*, vol. 1, no. 3-4, pp. 63–75, 2007.

[11] M. Orlov, "Comprehensive library of reusable semantically compatible components of next-generation intelligent computer systems," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*. Minsk : BSUIR, 2022, pp. 261–272.

[12] V. Golenkov, N. Guliakina, V. Golovko, V. Krasnoproshin, "Methodological problems of the current state of works in the field of artificial intelligence," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, ser. 5, V. Golenkov, Ed. BSUIR, Minsk, 2021, pp. 17–24.

[13] V. Ryen, A. Soylu, and D. Roman, "Building semantic knowledge graphs from (semi-) structured data: a review," *Future Internet*, vol. 14, no. 5, p. 129, 2022.

[14] P. Barnaghi, A. Sheth, and C. Henson, "From data to actionable knowledge: Big data challenges in the web of things [guest editors' introduction]," *IEEE Intelligent Systems*, vol. 28, no. 6, pp. 6–11, 2013.

[15] C. Kellogg, "From data management to knowledge management," *Computer*, vol. 19, no. 01, pp. 75–84, 1986.

[16] D. Shunkevich, "Universal model of interpreting logical-semantic models of intelligent computer systems of a new generation," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, V. Golenkov, Ed. BSUIR, Minsk, 2022, p. 285–296.

[17] Golenkov, V. V., "Graphodynamic models of parallel knowledge processing: principles of construction, implementation, and design," in *Open semantic technologies for designing intelligent systems (OSTIS-2012): Proceedings of the II International Scientific and Technical Conference, Minsk, February 16-18, 2012*, Belarusian State University of Informatics and Radioelectronics; editorial board: V. V. Golenkov (chief editor) [et al.]. Minsk, 2012, pp. 23–52.

[18] V. Ivashenko, "General-purpose semantic representation language and semantic space," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, ser. Iss. 6. Minsk : BSUIR, 2022, pp. 41–64.

[19] V. V. Golenkov and N. A. Gulyakina, "Structuring the semantic space," in *Open semantic technologies for designing intelligent systems (OSTIS-2014): Proceedings of the IV International Scientific and Technical Conference*, V. V. Golenkov, Ed. Minsk: BSUIR, 2 2014, pp. 65–78, chief editor Golenkov, V. V. [and others].

[20] ——, "Principles of building mass semantic technology for component design of intelligent systems," in *Open semantic technologies for designing intelligent systems (OSTIS-2011): Proceedings of the international scientific and technical conference*, V. V. Golenkov, Ed. Minsk: Belarusian State University of Informatics and Radioelectronics, 2 2011, pp. 21–58, chief editor Golenkov, V. V. [et al.].

[21] D. Shunkevich, "Hybrid problem solvers of intelligent computer systems of a new generation," *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*, no. 6, pp. 119–144, 2022.

[22] "Neo4j graph Database Platform | Graph Database Management System [Electronic resource]," April 2024. [Online]. Available: https://neo4j.com/

[23] T. Kahveci and A. K. Singh, "An efficient index structure for string databases," in *VLDB*, vol. 1, 2001, pp. 351–360.

[24] M. Barsky, U. Stege, and A. Thomo, "Structures for indexing substrings," in *Full-Text (Substring) Indexes in External Memory*. Springer, 2012, pp. 1–15.

[25] N. V. Zotov, "Model of process management in shared semantic memory of intelligent systems," in *Information Technologies and Systems 2023 (ITS 2023)*, L. Y. Shilin, Ed. Minsk: Belarusian State University of Informatics and Radioelectronics, 11 2023, pp. 53–54, proceedings of the International Scientific Conference.

[26] J. L. W. Kessels, "An alternative to event queues for synchronization in monitors," *Communications of the ACM*, vol. 20, no. 7, pp. 500–503, 1977.

[27] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[28] A. Zagorskiy, "Principles for implementing the ecosystem of next-generation intelligent computer systems," in *Otkrytye semanticheskie tekhnologii proektirovaniya intellektual'nykh system [Open semantic technologies for intelligent systems]*. BSUIR, Minsk, 2022, p. 347–356.

[29] R. Love, *Linux system programming: talking directly to the kernel and C library*. O'Reilly Media, Inc., 2013.

[30] "Software implementation of semantic networks processing storage [Electronic resource]," 2024, mode of access: https://github.com/ostis-ai/sc-machine. — Date of access: 30.03.2024.

[31] R. Bayer, "Prefix b-trees," *ACM Transactions on Database Systems (TODS)*, vol. 2, no. 1, pp. 11–26, 1977.

[32] P. Ferragina and R. Grossi, "The string b-tree: A new data structure for string search in external memory and its applications," *Journal of the ACM (JACM)*, vol. 46, no. 2, pp. 236–280, 1999.

[33] D. Belazzougui, "Fast prefix search in little space, with applications," in *European Symposium on Algorithms*, 2010, pp. 427–438.

[34] M. I. Cole, *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[35] L. Gonnord, L. Henrio, L. Morel, and G. Radanne, "A survey on parallelism and determinism," *ACM Computing Surveys*, vol. 55, no. 10, pp. 1–28, 2023.

[36] N. V. Zotov, "Quantitative indicators of operations efficiency over shared semantic memory of intelligent systems," in *Information Technologies and Systems 2023 (ITS 2023)*, L. Y. Shilin, Ed. Minsk: Belarusian State University of Informatics and Radioelectronics, 11 2023, pp. 51–52, proceedings of the International Scientific Conference.

[37] L. P. Miret, "Consistency models in modern distributed systems. an approach to eventual consistency," *Master. MA thesis. Universitat Politecnica de Valencia, Spain*, 2014.

ФОРМАЛЬНАЯ МОДЕЛЬ ОБЩЕЙ СЕМАНТИЧЕСКОЙ ПАМЯТИ ДЛЯ ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ НОВОГО ПОКОЛЕНИЯ

Зотов Н.В.

В работе подробно рассматривается формальная модель семантической памяти для интеллектуальных систем, структура, её элементы, соответствия между ними, правила и алгоритмы. Описывается реализация на основе данной модели, приводятся количественные показатели её эффективности.

Received 15.03.2024