Hybrid problem solvers of intelligent computer systems of a new generation

Daniil Shunkevich
Belarusian State University of Informatics and Radioelectronics
Minsk, Belarus
Email: shunkevich@bsuir.by

***Abstract*—In the article, the actual problems of the current state of technologies for the development of hybrid problem solvers are formulated, an approach to their solution based on the OSTIS Technology is proposed. The principles of building a problem solver as a hierarchical system of skills based on a multi-agent approach are formulated, the ontologies of agents and the actions performed by them are given. The principles of synchronization of agents' activities are formulated, as well as the ontology of the basic programming language for implementing agent programs and the interpreter model of such a language are developed.**

***Keywords*—OSTIS, problem solver, multi-agent system, problem-solving model, ontological approach**

I. Introduction

Currently, the usage of intelligent systems in a variety of fields is becoming increasingly relevant. One of the key components of an intelligent system that provides the ability to solve a wide range of problems is a problem solver. Its peculiarity in comparison with other modern software systems is the need to solve problems in conditions when the necessary information is not explicitly localized in the knowledge base of the intelligent system and must be found in the process of solving the problem, based on any criteria.

In other words, if in traditional systems, when solving a problem, it is always assumed that there are some localized source data ("given") and some description of the desired result ("what is required"), then in an intelligent system, all the information currently available in the system acts as source data when solving a large number of problems, that is, the entire knowledge base. In addition, if it is impossible to solve the problem in the current state of the knowledge base, the intelligent system should be able to understand what exactly is missing to continue the solution process and try to get the missing information in the external environment (for example, to request from the user).

To date, within various fields of artificial intelligence, a large number of different *problem-solving models* have been developed, each of which allows solving problems of a certain class. The expansion of the application fields for intelligent systems requires them to be able to solve so-called complex problems, the solution of each of which requires combining several problem-solving models, while it is not known a priori in what order and how many times one or another model will be used. Problem solvers, in which several problem-solving models are combined, are called *hybrid problem solvers*, and intelligent systems, in which various types of knowledge and various problem-solving models are combined, are called *hybrid intelligent systems* [1].

Improving the efficiency of the development and maintenance of hybrid intelligent systems requires the unification of models for the representation of various knowledge types and knowledge processing models, which would simplify the integration on its basis of components corresponding to different problem-solving models. Such models based on a unified semantic representation of information are proposed within an Open semantic technology for intelligent systems design (*OSTIS Technology*) [2] and are described within the corresponding *OSTIS Standard* [3]. In the article [2], the analysis of modern approaches to the development of hybrid problem solvers is carried out and it is shown that the approach proposed within the *OSTIS Technology* is currently the only example of a comprehensive approach to the development of hybrid problem solvers, within which the above problems are solved.

However, there are a number of problems that remain relevant and require solutions.

II. Current problems in the development of hybrid problem solvers

The first problem is related to the lack of a sufficiently strict formalized classification of problems solved by intelligent systems, the lack of unification in the description of problems and classes of problems, the description of purposes, progress, and result of solving the problem, problem-solving methods, relations between classes of problems and problem-solving methods of this class. The solution of this problem, on the one hand, will allow for the possibility of deep integration of various problem-solving models of various classes and the ability to simplify the process of integrating new problem-solving

models into an intelligent system and, on the other hand, will become a precondition for solving other problems described below.

The second problem is that at the moment the main attention in the field of developing hybrid problem solvers is paid to reducing the complexity of integrating various components of the problem solver into an intelligent system and realizing the possibility of accumulating reusable solvers components, but in general it is not said how specifically the intelligent system will use certain components in solving problems of specific classes. Thus, the creation of a general plan for solving a problem, i.e. the selection of problem-solving methods, the determination of the order of their application, and the choice of source data (arguments) for the usage of a particular method is actually determined by the developer at the stage of system design or its evolution during operation. The precondition for solving this problem is the solution of the previously considered problem of unifiying the representation of problems of various classes and methods for solving them. The solution of the problem under consideration involves the development of a set of *problem-solving strategies* (or problem-solving meta-methods) that will allow the intelligent system to independently form a plan for solving the problem, taking into account the problem-solving methods available in the system and, if possible, even request the missing components for solving the problem in the appropriate libraries. It should be noted that attempts to develop universal high-level approaches to solving problems were made at the dawn of the development of artificial intelligence, in the 1950s and 60s, but were unsuccessful and soon be abandoned. This is largely conditioned by the lack of unified models of knowledge representation and processing at that time, which are currently proposed within the *OSTIS Technology*.

Another urgent problem, closely related to those discussed above, is that intelligent systems are often forced to solve problems in the conditions of so-called non-factors, that is, when the description of the problem and possible ways to solve it are incomplete, the fuzziness and incorrectness of existing knowledge, as well as the lack of criteria for evaluating the optimality of the resulting solution, etc. take place [4]. This is especially relevant when solving behavioral problems related to changes in the state of objects of the environment external to the intelligent system. To solve problems in such conditions, an intelligent system must not only have a sufficient set of problem solver components that implement problem-solving models in the presence of non-factors (fuzzy logic models, machine learning models, genetic algorithms, etc.) but also implement *problem-solving strategies* that would allow making decisions and forming a plan for solving the problem in such conditions.

Problems considered are primarily related to the process of solving a specific problem by an intelligent system. At the same time, it is obvious that at every moment of time, an intelligent system is forced to solve several problems in parallel, which can be related both to the direct functional purpose of the system and to ensuring the operation and evolution of the system itself. In the second case, the problems related to updating the information it contains about the outside world, finding and eliminating errors in the knowledge base, optimizing the structure of the knowledge base and the solver of the system, finding and eliminating information garbage, and many others are meant. At the same time, different problems may have different priorities, which may vary depending on the situation, even in the process of its solution. At the same time, in a situation where it is not known a priori which of the possible ways to solve the problem will be the most effective, it may be advisable to use several approaches in parallel to solve the same problem. Thus, the problem of organizing the control of information processes for solving problems in an intelligent system and the interaction of information processes that occur in parallel, taking into account the priority of processes, the ability to monitor the current state of information processes, generate, suspend, and eliminate information processes is relevant. To solve this problem, it is advisable to borrow solutions widely used in traditional computer systems, in particular, implemented in modern operation systems, and adapt them to the specifics of solving problems in intelligent systems. It is important to note that the implementation of the information process control model, based on the general unified information processing models proposed within the OSTIS Technology, will make some information processes the object of analysis of other information processes, which, in turn, will make it possible to analyze the progress of solving the problem directly in the process of solving, evaluate the effectiveness of certain problem-solving methods, collect the most successful solutions for its further application in solution of the similar problems, and much more.

Solving these problems will allow developing a fundamentally new hierarchical model of a *hybrid problem solver*, which has a number of significant advantages, which, in turn, will have to be interpreted on any platforms. Without unifying the requirements for the platform of interpreting intelligent systems models and a clear separation of the platform-independent model of the system (and, in particular, the solver) and the platform, it is impossible to talk about the implementation of the solver model realizing the ideas discussed above. This will lead to the need to duplicate the same model components for different platforms and will significantly complicate the integration of solver components, since it will require taking into account the features of each platform during such integration. In addition, a clear separation of the

system model level and the platform level will make it possible to independently develop various platforms and models of intelligent systems. Thus, it is proposed to formulate unified requirements for the platform of interpreting semantic models of intelligent systems, as well as to build a general model of such a platform that meets these requirements.

On the other hand, as already mentioned, the problem solver is a complex system focused on working with knowledge, not with data, unlike modern software systems in which it is initially known where exactly the necessary data is localized and in what form they are represented. In this regard, the usage of modern hardware and software platforms, focused on address access to data stored in memory, for the development of intelligent systems is not always effective, since when developing intelligent systems, it is actually necessary to model nonlinear memory based on linear one. Increasing the efficiency of problem solving by intelligent systems requires the development of specialized platforms, including hardware ones, focused on unified semantic models of information representation and processing. As a basis for such developments, it is proposed to use the suggested within the *OSTIS Technology* general concepts of a semantic computer, semantic memory, and a basic programming language focused on processing information in such memory, and complement them with ideas of wave programming languages, insertion programming, and other approaches aimed at improving the efficiency of knowledge processing, including at the hardware level.

The development of problem solvers, including the problems of developing hybrid problem solvers discussed above, are currently being considered in the context of single (independent) intelligent systems operating in some environment (of which the user is also a part, if there is one). At the same time, there is an obvious tendency of modern information technologies to move from single systems to collectives of distributed interacting computer systems, in particular, to distributed data storage and distributed computing. In the case of intelligent computer systems, as the most important property of the systems included in such collectives, *interoperability* serves, that is, the ability of the system to coordinate interaction with other similar systems in order to solve any problems. Thus, the transition from the development of problem solvers of individual intelligent systems to problem solvers of interacting interoperable intelligent systems is particularly relevant, including the development of principles for solving problems in such distributed collectives, taking into account the solution of all the problems outlined above. To solve this problem, it is proposed to apply the ideas suggested within the theory of multi-agent systems and reinterpreted in the context of the interaction of hybrid intelligent systems.

In addition, the most important problem in the case of a distributed collective of intelligent systems is not just providing the ability to solve problems by such a collective at the current time but permanently supporting semantic compatibility and, as a consequence, the interoperability of systems included in such a collective throughout their entire life cycle. It is obvious that each of the systems included in such a collective and, accordingly, its problem solver can evolve independently of other systems, but at the same time, interoperability between systems must always be maintained, otherwise solving problems in such a collective will become impossible. The solution of this problem involves the development of methods for permanently analyzing semantic compatibility of a distributed collective of interacting intelligent systems, identification and elimination of problems.

Within this article, an approach to solving some of the listed problems based on the *OSTIS Technology* is proposed.

III. Proposed approach

As mentioned earlier, it is proposed to solve these problems within the *OSTIS Technology*. Let us list the basic principles of this technology that create preconditions for solving these problems:

- The *OSTIS Technology* is based on a universal method of semantic representation (encoding) of information in the memory of intelligent computer systems, called an *SC-code*. Texts of the *SC-code* (sc-texts, sc-constructions) 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, depending on orientation, can be *sc-arcs* or *sc-edges*). The *Alphabet of the SC-code* consists of five main elements, on the basis of which SC-code constructions of any complexity are built, including more specific types of sc-elements (for example, new concepts). The universality and uniformity of the *SC-code* makes it possible to describe on its basis any *knowledge types* and any problem-solving *methods*, which, in turn, greatly simplifies their integration within one system. Systems developed based on the *OSTIS Technology* are called *ostis-systems*;
- The basis of the knowledge base developed by the *OSTIS Technology* is a hierarchical system of semantic models of *subject domains* and *ontologies*, among which the universal *Kernel of the knowledge base semantic models* and the methodology for the development of semantic knowledge base models are allocated, which ensure the semantic compatibility of the knowledge bases being developed;
- The basis of information processing within the *OSTIS Technology* is the *SCP Language*, the program texts of which are also written in the form of SC-code constructions;

- The problem solver architecture within the *OSTIS Technology* is based on a multi-agent approach, in which agents interact with each other purely by specifying the actions they perform within a common semantic memory (such agents are called *sc-agents*). Such an approach allows ensuring the fundamental possibility of implementing any *problem-solving methods* in the form of corresponding solver components and providing their semantic compatibility. Other advantages of the multi-agent approach in general are widely known and discussed in related publications [5]–[7].

The listed principles of the *OSTIS Techology* are proposed to be supplemented with some of the ideas underlying the solution of those problems and, taking this into account, to develop:

- A complex ontology of actions, problems, and methods of their solution, as well as an ontology of *hybrid problem solvers*, on the basis of which to clarify the concept of the solver and its architecture. The first version of the *Global subject domain of actions and problems and the corresponding ontology of methods and technologies* is already represented within the *OSTIS Standard*, on its basis it is proposed to develop an ontology of actions and problems solved by *ostis-systems*;
- A complex of unified generalized strategies (meta-methods) for solving problems in intelligent systems, which allows an intelligent system to independently form a plan for solving a problem, taking into account the problem-solving methods available in the system. In addition to the experience of similar works, it is also proposed to supplement the developed strategies with some general methodological ideas related to the theory of behaviorism and the ideas of its application in computer science that are gaining popularity [8]–[10], TIPS [11], as well as the STA-methodology proposed by the school of G. Shchedrovitsky [12];
- An ontological model for the formation of a plan for solving a problem and managing the process of solving problems in hybrid problem solvers under conditions of various non-factors and the absence of clear criteria for evaluating the optimality of the resulting solution. To develop this model, it is proposed to adapt the theory of situational control proposed by D. Pospelov [13] and implement it in the context of the semantic theory of problem solvers developed within the OSTIS Technology;
- An ontological model for controlling information processes for solving problems in intelligent systems built on the basis of unified semantic models of information representation and processing;
- An ontological model of the platform for interpreting unified semantic models of information representation and processing (*ostis-platforms*);
- A comprehensive hierarchical model of a hybrid problem solver based on a multi-agent approach and taking into account the need to solve problems both within single intelligent systems and within distributed collectives of interoperable intelligent systems;
- A complex of methods for analyzing the quality of hybrid problem solvers and their components;
- A complex of tools to support the design of hybrid problem solvers.

Within the *OSTIS Technology*, several universal variants of visualization of *SC-code* constructions are proposed, such as *SCg-code* (graphic variant), *SCn-code* (nonlinear hypertext variant), *SCs-code* (linear string variant). Within this article, fragments of structured texts in the SCn code [3] will often be used, which are simultaneously fragments of the source texts of the knowledge base, understandable to both human and machine. This allows making the text more structured and formalized, while maintaining its readability. The symbol "**:=**" in such texts indicates alternative (synonymous) names of the described entity, revealing in more detail certain of its features.

As follows from the principles of the *OSTIS Technology* discussed earlier, the building of ontological models of any entities involves the development of an appropriate *subject domain and ontology* (or a family of *subject domains and ontologies*), within which the properties of this entity are clarified by a formal description of the corresponding set of concepts, including relations.

Within this work, we will consider in more detail the fragments of:

- The *Subject domain and ontology of ostis-systems problem solvers*, which clarifies the concepts of a problem solver, a knowledge processing machine, as well as the classification of problem solvers and knowledge processing machines;
- The *Subject domain and ontology of actions and problems of ostis-systems*, within which the classes of actions and problems solved in ostis-systems are specified;
- The *Subject domain and ontology of sc-agents*, which clarifies the concept of an sc-agent as a component of the ostis-system problem solver, the typology of sc-agents and their properties, as well as the principles for synchronizing the activities of sc-agents;
- The *Subject domain and ontology of the Basic programming language of ostis-systems*, which clarifies the syntax, denotational semantics, and operational semantics of the SCP Language, which is the basic for ostis-systems.

IV. Ostis-systems problem solvers

Within the *OSTIS Technology*, the *ostis-system problem solver* is defined as the totality of all *skills* possessed by

the ostis-system at the current time [3], [14].

In turn, the skill is interpreted as a combination of some *method* and its operational semantics, that is, information about how this *method* should be interpreted.

By a *method* we will understand the description of how any or almost any (with explicit exceptions) action belonging to the corresponding *action class* can be performed. Since a specific *action class* corresponds to some specific *problem class*, we can say that the method describes a way to solve any problems belonging to a given class. The concept of a method can be considered as a generalization of the concept of "program", in connection with which, within the OSTIS Technology, the terms "method" and "program" are synonymous [14].

As an example of a particular method, a procedural program in a specific programming language or a set of logical propositions that make up a formal theory of a given subject domain (analogous to a logical program) can be used.

A particular case of the method is the program of the atomic component of the *ostis-system problem solver* (*atomic sc-agent*); in this case, a collective of lower-level agents, interpreting the corresponding program, acts as the operational semantics of the method (in the extreme case, these will be agents that are part of the platform for interpreting computer system models, including the hardware one).

Thus, we can talk about the hierarchy of *methods* and *methods* for interpreting other *methods*. Taking into account this thesis, it is possible to clarify the concept of a problem solver as a hierarchical system of skills.

An approach to the building of problem solvers proposed within the *OSTIS Technology* allows them to be modifiable, which, in turn, allows the *ostis-system*, if necessary, to easily acquire new *skills*, modify (improve) existing ones, and even get rid of some skills in order to improve system performance. Thus, it makes sense to talk not about a rigidly fixed problem solver, which is developed once when creating the first version of the system and does not change further, but about a set of skills fixed at each current moment of time but constantly evolving.

ostis-system problem solver
⇐ *family of subsets**:
  skill
:= [hierarchical system of skills possessed by the ostis-system]
⊃ *hybrid ostis-system problem solver*
  := [ostis-system problem solver that implements two or more problem-solving models]
⊃ *combined ostis-system problem solver*
  := [complete ostis-system problem solver]
  := [integrated ostis-system problem solver]
  := [ostis-system problem solver that implements all its functionality, both basic and auxiliary]

In general, the *combined ostis-system problem solver* solves problems related to:

- providing the basic functionality of the system (for example, solving explicitly formulated problems at the user's request);
- ensuring the correctness and optimization of the ostis-system itself (permanently throughout the entire life cycle of the ostis-system);
- providing advanced training for end users and developers of the ostis-system;
- providing automation of the design and control of the development of the ostis-system.

By a *knowledge processing machine* we will understand the set of interpreters of all *skills* that make up some *problem solver*. Taking into account the multi-agent approach to information processing used within the OSTIS Technology, the *knowledge processing machine* is an *sc-agent* (most often – a *non-atomic sc-agent*), which includes simpler sc-agents that provide interpretation of the corresponding set of *methods*. Thus, the *knowledge processing machine* in general is a hierarchical system of *sc-agents*.

Taking into account the fact that there is a hierarchy of methods in terms of the level of interpretation (some methods interpret others), it is also necessary to talk about the hierarchy of *knowledge processing machines*.

knowledge processing machine
⊂ *sc-agent*

Let us consider the classification of ostis-systems problem solvers according to various criteria.

Classification of ostis-systems problem solvers by the type of the corresponding ostis-system:

ostis-system problem solver
∋ *Problem solver of the IMS.ostis Metasystem*
⊃ *problem solver of the auxiliary ostis-system*
  ⊃ *problem solver of the computer system interface*
  ⊃ *ostis-subsystem problem solver for supporting the design of components of a certain class*
    ⊃ *ostis-subsystem problem solver for supporting knowledge base design*
    ⊃ *ostis-subsystem problem solver for supporting the design of ostis-systems problem solvers*
  ⊃ *problem solver of the control subsystem for the design of computer systems and their components*

⊃ *problem solver of an independent ostis-system*

problem solver of the computer system interface
⇒ *subdividing**:
{• *problem solver of the user interface of a computer system*
• *problem solver of the computer system interface with other computer systems*
• *problem solver of the computer system interface with the environment*
}

ostis-subsystem problem solver for supporting knowledge base design
⊃ *problem solver for improving the quality of the knowledge base*
⊃ *problem solver for knowledge base verification*
⊃ *problem solver for finding and eliminating inaccuracies in the knowledge base*
⊃ *problem solver for finding and eliminating incompleteness*
⊃ *problem solver for optimizing the structure of the knowledge base*
⊃ *problem solver for identifying and eliminating information garbage*

ostis-subsystem problem solver for supporting the design of ostis-systems problem solvers
⇒ *subdividing**:
{• *ostis-subsystem problem solver for supporting the design of knowledge processing programs*
• *ostis-subsystem problem solver for supporting the design of knowledge processing agents*
}

Classification of ostis-systems problem solvers by the type of interpreted problem-solving model:

ostis-system problem solver
⊃ *problem solver with stored methods*
:= [solver capable of solving problems of those classes for which the corresponding solution method is known at a given moment]
⊃ *problem solver based on neural network models*
⊃ *problem solver based on genetic algorithms*
⊃ *problem solver based on imperative programs*
⊃ *problem solver based on procedural programs*
⊃ *problem solver based on object-oriented programs*
⊃ *problem solver based on declarative programs*
⊃ *problem solver based on logical programs*
⊃ *problem solver based on functional programs*
⊃ *problem solver in conditions when the method of solving problems of this class is not known at the current time*
:= [solver that implements problem-solving strategies that allow generating a problem-solving method that is not currently known to the ostis-system]
:= [solver that uses meta-methods for solving problems, corresponding to more general classes of problems in relation to a given one]
:= [problem solver that allows generating a method that is particular in relation to any method known to the ostis-system and is interpreted by the corresponding knowledge processing machine]
⊃ *solver that implements the strategy of finding ways to solve the problem in depth*
⊃ *solver that implements a strategy for finding ways to solve a problem in width*
⊃ *solver that implements a trial-and-error strategy*
⊃ *solver that implements a strategy for splitting a problem into subproblems*
⊃ *solver that implements a strategy for solving problems by analogy*
⊃ *solver that implements a concept of an intelligent software package*

Separately, we will highlight the classification of knowledge processing machines, which in general can correspond to the same fragments of the knowledge base but together with them form different skills and, accordingly, different problem solvers:

knowledge processing machine
⊃ *logical inference machine*
⊃ *deductive inference machine*
⊃ *direct deductive inference machine*
⊃ *reverse deductive inference machine*
⊃ *inductive inference machine*
⊃ *abductive inference machine*
⊃ *fuzzy inference machine*
⊃ *inference machine based on default logic*
⊃ *logical inference machine with*

consideration for the time factor

Classification of ostis-systems problem solvers by the type of problem to be solved (purposes of solving the problem):

ostis-system problem solver
- ⊃ *problem solver for information search*
  - ⇒ *subdividing**:
    - {• *problem solver for finding information that meets the specified criteria*
    - • *problem solver for finding information that does not meet the specified criteria*
    - }
- ⊃ *solver of explicitly formulated problems*
  - := [problem solver for which the purpose is explicitly formulated]
  - ⊃ *problem solver for searching or calculating the values of a given set of quantities*
  - ⊃ *problem solver for establishing the truth of a given logical proposition within a given formal theory*
  - ⊃ *problem solver for forming a proof of a given proposition within a given formal theory*
  - ⊃ *machine for verifying the response to the specified problem*
  - ⊃ *machine for verifying the solution of the specified problem*
    - ⊃ *machine for verifying the proof of a given proposition within a given formal theory*
- ⊃ *problem solver for entity classification*
  - ⊃ *machine for correlating an entity with one of a given set of classes*
  - ⊃ *machine for dividing a set of entities into classes according to a given set of attributes*
- ⊃ *problem solver for the synthesis of information constructions*
  - ⊃ *problem solver for the synthesis of natural language texts*
  - ⊃ *problem solver for image synthesis*
  - ⊃ *problem solver for signal synthesis*
    - ⊃ *problem solver for speech synthesis*
- ⊃ *problem solver for the analysis of information constructions*
  - ⊃ *problem solver for analysis of natural language texts*
    - ⊃ *problem solver for understanding natural language texts*
    - ⊃ *problem solver for verification of natural language texts*
  - ⊃ *problem solver for image analysis*
    - ⊃ *problem solver for image segmentation*
    - ⊃ *problem solver for understanding images*
  - ⊃ *problem solver for signal analysis*
    - ⊃ *problem solver for speech analysis*
      - ⊃ *problem solver of speech understanding*

V. General principles of information processing in ostis-systems

The proposed approach to problem solving is based on a number of ideas related to the concept of situational control proposed in the work of D. Pospelov [13]. To date, attempts to implement this concept, despite its relevance and demand, have been reduced to particular solutions for specific classes of problems and, unfortunately, have not been widely distributed. To a large extent, this is conditioned by the lack of a universal unified basis that would make it possible to create situational control languages based on it in application to specific subject domains and, more importantly, reuse fragments of descriptions in such languages.

This problem can be solved using an *SC-code*, proposed within the *OSTIS Technology*, and a family of top-level ontologies developed on its basis. In particular, the implementation of the ideas of situational control is facilitated by such principles as:

- the SC-code as a basic language for describing any information in the knowledge base and, accordingly, for building situational control languages based on it;
- basic set-theoretic semantics of the SC-code, which makes it possible to formally clarify all the concepts used in the form of a formal set of ontologies, which allows for compatibility of the systems being developed and the possibility of reusage of their components;
- an agent-oriented approach to information processing, involving the reaction of agents to the occurrence of certain situations and events in the knowledge base.

Let us consider in more detail the basic principles of information processing underlying the proposed approach:

- The problem solver of each *ostis-system* is based on a multi-agent system whose agents interact with each other only(!) through their shared *sc-memory* by specifying in this memory the *actions in sc-memory* performed by them. At the same time, users of the *ostis-system* are also considered as agents of this system. In addition, *sc-agents* are divided into internal, receptor, and effector. Interaction between

agents via shared *sc-memory* is reduced to the following types of actions:

1) usage of the part of the stored knowledge base that is available for the corresponding group of sc-agents;
2) formation (generation) of new fragments of the knowledge base and/or correction (editing) of any fragments of the available part of the knowledge base;
3) integration (immersion) of new and/or updated fragments into the available part of the knowledge base.

Let us emphasize that sc-agents do not communicate with each other directly by sending messages, as is done in most modern approaches to building multi-agent systems. In addition, sc-agents have access to a common knowledge base, which guarantees semantic compatibility (mutual understanding) between agents, including users of ostis-systems.

- The user of the *ostis-system* cannot directly perform any action in sc-memory, but via the user interface they can initiate the construction (generation, formation in *sc-memory*) of *sc-text*, which is a specification of the *action in sc-memory* performed either by one *atomic sc-agent* in one act, or by one *atomic sc-agent* in several acts, or by a collective of *sc-agents* (*non-atomic sc-agent*). In the specification of each such *action in sc-memory* initiated by a user, this user is indicated as the customer of this action. Thus, the user of the *ostis-system* gives instructions (tasks, commands) to *sc-agents* of this system to perform various actions specified by them in *sc-memory*.
- Each *sc-agent*, performing some *action in sc-memory*, have to "remember" that *sc-memory*, on which it is working, is a shared resource not only for it but also for all others *sc-agents*, working on the same *sc-memory*, therefore, the *sc-agent* must comply with a certain ethics for behaving in a collective of such *sc-agents*, which should minimize the interferences that it creates to other *sc-agents*.
- The activity of each agent of the *ostis-system* is discrete and represents a set of elementary actions (acts). At the same time, when performing each act, the agent can set several types of locks on fragments of the knowledge base. These locks allow prohibiting other agents from changing the specified fragment of the knowledge base or even making it "invisible" to other agents. The locks are set by the agent itself during the execution of the relevant act and are removed by it at the last stage of the execution of this act or earlier, if possible.
- If a certain *sc-agent* performs some *action in sc-memory*, then, for the duration of this action, it can:

1) prohibit other *sc-agents* from changing the state of some sc-elements stored in *sc-memory* – delete them, change the type;
2) prohibit other *sc-agents* from adding or deleting elements of some sets denoted by the corresponding *sc-nodes*;
3) prohibit other *sc-agents* from viewing some *sc-elements*, that is, these *sc-elements* become completely "invisible" (completely blocked) for other *sc-agents* but only for the duration of performing the proper action.

The specified locks must be completely removed before the completion of the corresponding action. Let us emphasize that the number of *sc-elements* blocked for the duration of some action mainly includes atomic and non-atomic connectives and should not include *sc-nodes* denoting infinite classes of any entities and, moreover, sc-nodes denoting various concepts (key classes of various subject domains).

Ethical (non-selfish) behavior of the *sc-agent* concerning blocking of *sc-elements* (that is, restricting access to them to other *sc-agents*) implies compliance with the following rules:

1) there should not be more *sc-elements* blocked than is necessary to solve the problem;
2) as soon as for any *sc-element* the need to lock it disappears before the completion of the corresponding action, it is advisable to immediately unlock this *sc-element* (remove the lock).

In order for the *sc-agent* to be able to work with any random *sc-element*, it must either make sure that this *sc-element* is not included in the knowledge base fragment that is part of the *full lock* or make sure that this lock is not set by this agent.

A special group of completely blocked *sc-elements* (for the duration of the action by the *sc-agent*) are auxiliary *sc-elements* ("scaffolds"), created only for the duration of this action. These sc-elements should not be unblocked at the end of the action but need to be deleted).

- If an *action in sc-memory* performed by the *sc-agent* has completed (i.e. has become a past entity), then the *sc-agent* registers the result of this *action*, specifying (1) deleted *sc-elements* and generated sc-elements. This is necessary if for some reason it will be required to rollback this *action*, i.e. to return to the state of the knowledge base before performing the specified *action*.

Let us list some advantages of the proposed approach to the organization of knowledge processing in ostis-systems:

- since processing is carried out by agents that exchange messages only through shared memory, adding a new agent or excluding (deactivating) one or more existing agents usually does not lead to

changes in other agents, since agents do not exchange messages directly;
- agent initiation is carried out in a decentralized manner and most often independently of each other, so even a significant expansion of the number of agents within one system does not lead to a deterioration in its performance;
- agent specifications and, as will be shown below, their programs can be written in the same language as the processed knowledge, which significantly reduces the list of specialized tools developed for the design of such agents and their collectives, as well as their analysis, verification, and optimization, and simplifies the development of the system by using more universal components.

VI. Actions and problems in ostis-systems

The building problem solvers and their components implies the need to describe the actions they perform and the problems they solve.

A. Concept of action in sc-memory

action in sc-memory
:= [internal action of the ostis-system]
:= [action performed in sc-memory]
:= [action performed in an abstract unified semantic memory]
:= [action performed by the ostis-system knowledge processing machine]
:= [action performed by an agent or a collective of agents of the ostis-system]
:= [information process on the knowledge base stored in sc-memory]
:= [process of solving an information problem in sc-memory]
⊂ *process in sc-memory*

Each ***action in sc-memory*** denotes some transformation performed by some *sc-agent* (or a collective of *sc-agents*) and focused on the transformation of *sc-memory*. The specification of the action after its execution can be included in the protocol for solving some problem.

The transformation of the state of the knowledge base includes, among other things, information search, which assumes (1) localization of the response to the request in the knowledge base, explicit allocation of the response structure, and (2) translation of the response into some external language.

The set of ***actions in sc-memory*** includes signs of actions of various kinds, the semantics of each of which depends on the specific context, i.e. the orientation of the action to any specific objects and the belonging of the action to any particular class of actions.

It should be clearly distinguished:
- each specific ***action in sc-memory***, which is some kind of transition process that transfers sc-memory from one state to another;
- each type of ***actions in sc-memory***, which is a certain class of similar actions (in one sense or another);
- sc-node denoting some specific ***action in sc-memory***;
- sc-node denoting a structure that is a description, specification, task, statement of the corresponding action.

Let us consider in more detail the classification of actions in sc-memory:

action in sc-memory
⊃ *action in sc-memory initiated by a question*
⊃ *action of editing the ostis-system knowledge base*
⊃ *action of setting the ostis-system mode*
⊃ *action of editing a file stored in sc-memory*
⊃ *action of interpreting a program stored in sc-memory*
  ⊃ *action of scp-program interpretation*

action in sc-memory initiated by a question
:= [action aimed at forming an answer to the question posed]
⊃ *action. create the specified file*
⊃ *action. create the specified structure*
  ⊃ *action. verify the specified structure*
    ⊃ *action. determine the truth or falsity of the indicated logical proposition*
    ⊃ *action. determine the correctness or incorrectness of the specified structure*
    ⊃ *action. create a structure describing the inaccuracies that exist in the specified structure*
  ⊃ *action. clarify the type of the specified sc-element*
    ⊃ *action. determine the positivity/negativity of the indicated sc-arc of belonging or non-belonging*
  ⊃ *action. create a semantic neighborhood*
    ⊃ *action. create a complete semantic neighborhood of the specified entity*
    ⊃ *action. create a basic semantic neighborhood of the specified entity*
    ⊃ *action. create a particular semantic neighborhood of the specified entity*
  ⊃ *action. create a structure describing the relations between the specified entities*
    ⊃ *action. create a structure*

describing the similarities of the specified entities
⊃ *action. create a structure describing the differences of the specified entities*
⊃ *action. create a structure describing the way to solve the specified problem*
⊃ *action. create a plan for generating an answer to the specified question*
⊃ *action. create a protocol for performing the specified action*
⊃ *action. create a justification for the correctness of the indicated solution*
⊃ *action. verify the justification of the correctness of the specified solution*
⊃ *action aimed at establishing the temporal characteristics of the specified entity*
⊃ *action aimed at establishing the spatial characteristics of the specified entity*

action of editing the knowledge base
⊃ *action. change the direction of the specified sc-arc*
⊃ *action. fix errors in the specified structure*
⊃ *action. transform the specified structure according to the specified rule*
⊃ *action. equate two specified sc-elements*
⊃ *action. include a set*
:= [make all elements of the ***Si*** set explicitly belonging to the ***Sj*** set, that is, generate the corresponding sc-arcs of belonging]
⊃ *action of generating sc-elements*
⊃ *action of generation, one of the arguments of which is some generalized structure*
⊃ *action. generate a structure isomorphic to the specified template*
⊃ *action. generate an sc-element of the specified type*
⊃ *action. generate an sc-connector of the specified type*
⊃ *action. generate an sc-node of the specified type*
⊃ *action. generate a file with the specified contents*
⊃ *action. set the specified file as the primary identifier of the specified sc-element for the specified external language*
⊃ *action. update concepts*
:= [action. replace non-basic concepts with their definition through basic concepts]
:= [action. replace some set of concepts with another set of concepts]
⊃ *action. integrate the information construction into the current state of the knowledge base*
⊃ *action. integrate the contents of the specified file into the current state of the knowledge base*
⊃ *action. translate the contents of the specified file to sc-memory*
⊃ *action. integrate the specified structure into the current state of the knowledge base*
⊃ *action. supplement the description of the past state of the ostis-system*
⊃ *action. supplement the structure describing the history of the ostis-system evolution*
⊃ *action. supplement the structure describing the history of ostis-system operation*
⊃ *action of deleting sc-elements*
⊃ *action. delete the specified sc-elements*
⊃ *action. delete sc-elements that are part of the specified structure and are not the key nodes of any sc-agents*

action. equate two specified sc-elements
:= [action. combine two specified sc-elements]
:= [action. paste two specified sc-elements together]
⇒ *subdividing**:
{• *action. physically equate two specified sc-elements*
• *action. logically equate two specified sc-elements*
}

Each ***action. equate two specified sc-elements*** can be performed as *action. physically equate two specified sc-elements* or *action. logically equate two specified sc-elements*. In the case of logical equation, the action itself is saved in the agent activity protocol with its specification, which includes a necessary indication of which elements were generated and which were deleted. In the case of physical equation, the action protocol is not saved.

Each ***action. update concepts*** denotes the transition from some group of concepts used earlier to another group of concepts that will be used instead of the first ones and will become *basic concepts*. In general, ***action. update concepts*** consists of the following steps:
- determine the concepts to be replaced based on the substitutive ones;
- make appropriate changes to the programs of sc-agents, the key nodes of which are updated concepts;
- replace all constructions in the knowledge base containing replaceable concepts, in accordance with the definitions of these concepts through the concepts that replace them;
- if necessary, *sc-elements* denoting the concepts replaced in this way can be completely deleted from

the current state of the knowledge base.

The first argument (included in the *action* sign under attribute *1'*) of ***action. update concepts*** is a sign for the set of *sc-nodes* denoting the replaced concepts, the second one (included in the *action* sign under attribute *2'*) is a sign for the set of *sc-nodes* denoting the replacing concepts. In general, either or both of these sets can be *singletons*.

action. delete the specified sc-elements
⇒ *subdividing**:
{• *action. physically delete the specified sc-elements*
• *action. logically delete the specified sc-elements*
}

Each ***action. delete the specified sc-elements*** can be performed as *action. physically delete the specified sc-elements* or *action. logically delete the specified sc-elements*. In the case of logical deletion, the action itself is saved in the agent activity protocol with its specification, which includes a necessary indication of which elements were deleted, i.e., in fact, the elements are excluded from the current state of the knowledge base. In case of physical deletion, the action protocol is not saved.

If any *sc-element* is deleted, the incident *connectives*, including *sc-connectors*, are also deleted.

To perform ***action. integrate the specified structure into the current state of the knowledge base***, it is necessary to paste *sc-elements* included in the integrated *structure* together with synonymous *sc-elements* included in the current state of the knowledge base, replace unused (for example, outdated) concepts included in the integrated *structure* on used ones (i.e. replace unused concepts with their definitions through used ones), explicitly include all elements of the integrated *structure* in the number of elements of the approved part of the knowledge base, and explicitly include all elements of the integrated *structure* in the number of elements that are part of any atomic sections of the approved fragment of the knowledge base.

B. Problems solved in sc-memory and logically atomic actions

problem solved in sc-memory
⊂ *problem*
:= [specification of the action performed in sc-memory]
:= [structure that is such a description (formulation, setting) of the corresponding action in sc-memory, which has sufficient completeness to perform the specified action]
:= [semantic neighborhood of some action in sc-memory, providing a sufficiently complete setting of this action]

action class
⊃ *action class in sc-memory*
⇐ *family of subsets**:
action in sc-memory
⇒ *subdividing**:
{• *class of logically atomic actions*
:= [class of autonomous actions]
⊃ *class of logically atomic actions in sc-memory*
• *class of logically non-atomic actions*
:= [class of non-autonomous actions]
}

Each *action* belonging to some specific *class of logically atomic actions* has two necessary properties:

- the execution of an action does not depend on whether the specified action is part of the decomposition of a more general action. When performing this action, the fact that this action precedes or follows any other actions should also not be taken into account (which is explicitly indicated using the *sequence of actions** relation);
- the specified action should be a logically integral act of transformation, for example, in semantic memory. Such an action is essentially a transaction, i.e. the result of such a transformation is a new state of the system being transformed, and the action being performed must either be performed completely or not at all, partial execution is not allowed.

At the same time, logical atomicity does not prohibit decomposing the performed action into more particular ones, each of which, in turn, will also be logically atomic.

It is proposed to divide all activities aimed at solving any problems by the ostis-system into logically atomic actions. This approach will allow for the modifiability of *ostis-systems problem solvers*, provided that the solver components correspond to *classes of logically atomic actions in sc-memory*. Such components are called sc-agents.

VII. Concept of an sc-agent and abstract sc-agent

sc-agent
:= [the only kind of *subjects* performing transformations in *sc-memory*]
:= [*subject* capable of performing *actions in sc-memory*, belonging to some specific *class of logically atomic actions*]

The logical atomicity of the actions performed by the sc-agent assumes that each sc-agent reacts to the corresponding class of situations and/or events occurring in the sc-memory and performs a certain transformation of the sc-text located in the semantic neighborhood of the processed situation and/or event. At the same time, each

sc-agent generally does not contain information about which other sc-agents are currently present in the system and interacts with other sc-agents solely by forming some constructions (usually action specifications) in the shared sc-memory. As such a message, for example, a question addressed to other sc-agents in the system (it is not known in advance which one specifically) or an answer to a question posed by other sc-agents (it is not known in advance which one specifically) can serve. Thus, each sc-agent at any given time controls only a fragment of the knowledge base in the context of the problem being solved by this agent; the state of the rest of the knowledge base is generally unpredictable for the sc-agent.

Since it is assumed that copies of the same *sc-agent* or functionally equivalent *sc-agents* can work in different ostis-systems, while being physically different sc-agents, it is advisable to consider the properties and classification of non-sc-agents but classes of functionally equivalent sc-agents, which we will call ***abstract sc-agents***. Under the ***abstract sc-agent*** is understood a certain class of functionally equivalent *sc-agents*, different instances (i.e. representatives) of which can be implemented in different ways.

Each ***abstract sc-agent*** has a corresponding specification. The specification of each ***abstract sc-agent*** includes:

- specifying the key *sc-elements* of this *sc-agent*, i.e. those *sc-elements* stored in *sc-memory* that are "support points" for this *sc-agent*;
- a formal description of the conditions for initiating this *sc-agent*, i.e. those *situation* in *sc-memory* that initiate the activity of this *sc-agent*;
- a formal description of the primary initiation condition for this *sc-agent*, i.e. such a situation in *sc-memory*, which prompts the *sc-agent* to switch to the active state and start checking for its full initiation condition (for *internal abstract sc-agents*);
- a strict, complete, unambiguously understood description of the activity of this *sc-agent*, drawn up using any understandable, generally accepted means that do not require special study, for example, in natural language;
- a description of the results of executing this *sc-agent*.

Sc-agents can be classified according to various criteria. Since we can talk about a hierarchy of methods (methods of interpreting other methods) and, accordingly, a hierarchy of skills, there is a need to talk about a hierarchy of sc-agents providing interpretation of a particular method. In this context, we can talk about the hierarchy of sc-agents in two aspects:

- an *abstract sc-agent* (and, accordingly, an *sc-agent*) can uniquely correspond to a *method* (sc-agent program) describing the activity of this sc-agent. Such agents will be called *atomic abstract sc-agents*;
- sometimes, it is advisable to combine *abstract sc-agents* into collectives of such agents, which can be considered as one integral *abstract sc-agent*, from a logical point of view, working on the same principles as *atomic abstract sc-agents*, that is, reacting to events in sc-memory and describing its activities within this memory. Such an *abstract sc-agent* will not correspond to any specific *method* stored in sc-memory, but the rest of the specification of the *abstract sc-agent* (initiation condition, initial situation description, and the result of the operation of the sc-agent, etc.) remains the same, like in case of the *atomic abstract sc-agent*. Thus, we can say that the concept of atomicity/non-atomicity of an abstract sc-agent indicates how the implementation of this *abstract sc-agent* is refined – by specifying a particular method (sc-agent program) or by decomposing the *abstract sc-agent* into simpler ones. It is important to note that *non-atomic abstract sc-agents* can also be part of other, more complex *non-atomic abstract sc-agents*. Thus, a hierarchical system of abstract sc-agents is formed, in general, having a random number of levels.
- In turn, the method corresponding to the sc-agent must be interpreted by some other sc-agent of a lower level and most often by a collective of such agents, each of which is assigned its own method describing the behavior of this agent but at a lower level. Thus, we can say that the concept of atomicity/non-atomicity of abstract sc-agents is applicable within one *method description language*. In turn, we can talk about the hierarchy of *abstract sc-agents* from the point of view of the language level for description of the methods corresponding to such agents. In general, such a hierarchy can also have an unlimited number of levels, however, it is obvious that when lowering the level of the method description language, sooner or later we must approach the method description language, which will be interpreted by agents implemented at the level of the *ostis-platform*, and going even lower – to the level of the method description language, interpreted at the hardware level. Thus, in order to ensure the platform independence of *ostis-systems*, it is advisable to allocate a method description language that would be interpreted at the level of the *ostis-platform* and be the basis for the development of interpreters of higher-level languages. As such a language, an *SCP Language* (Semantic Code Programming) is proposed, which is considered as an assembler for an *associative semantic computer*.

The hierarchical approach to the description of *knowledge processing machines* and, accordingly, *problem solvers* has a number of important advantages, such as ensuring the modifiability of solvers and the convenience of their design and debugging at different levels [2], [3].

Let us consider the classification of *abstract sc-agents* according to various criteria. Classification of *abstract sc-agents* based on atomicity:

abstract sc-agent
⇒ *subdividing**:
{• *non-atomic abstract sc-agent*
• *atomic abstract sc-agent*
}

A ***non-atomic abstract sc-agent*** is understood as an *abstract sc-agent*, which is decomposed into a collective of simpler *abstract sc-agents*, each of which in turn can be both an *atomic abstract sc-agent* and ***non-atomic abstract sc-agent***. At the same time, in some variant of *decomposition of an abstract sc-agent**, the child ***non-atomic abstract sc-agent*** can become an *atomic abstract sc-agent* and be implemented accordingly.

An ***atomic abstract sc-agent*** is understood as an *abstract sc-agent*, for which the method of its implementation is specified, i.e. there is a corresponding connective of the *sc-agent program** relation.

The *SCP Language* allows setting boundaries between the logical-semantic model of the *ostis-system* and the *ostis-platform*. In this regard, we will consider *abstract sc-agents* as platform-independent ones, implemented in the SCP Language or higher-level languages based on it, and *abstract sc-agents* – as platform-dependent ones, that are implemented at the platform level (for example, in order to improve their performance). At the same time, there are a number of abstract sc-agents that cannot be implemented in principle in the *SCP Language*. This is represented in the following hierarchy:

abstract sc-agent
⇒ *subdividing**:
{• *internal abstract sc-agent*
• *effector abstract sc-agent*
• *receptor abstract sc-agent*
}
⇒ *subdividing**:
{• *abstract sc-agent that is not implemented in the SCP Language*
• *abstract sc-agent that is implemented in the SCP Language*
}
⇒ *subdividing**:
{• *abstract sc-agent for interpreting scp-programs*
• *abstract software sc-agent*
• *abstract sc-meta-agent*
}
⇒ *subdividing**:
{• *platform-dependent abstract sc-agent*
⊃ *abstract sc-agent that is not implemented in the SCP Language*
• *platform-independent abstract sc-agent*
}

abstract sc-agent that is not implemented in the SCP Language
:= [abstract sc-agent that cannot be implemented at a platform-independent level]
⇒ *subdividing**:
{• *effector abstract sc-agent*
• *receptor abstract sc-agent*
• *abstract sc-agent for interpreting scp-programs*
}

abstract sc-agent that is implemented in the SCP Language
:= [abstract sc-agent that can be implemented at a platform-independent level]
⇒ *subdividing**:
{• *abstract sc-meta-agent*
• *abstract software sc-agent implemented in the SCP Language*
}

abstract software sc-agent
⇒ *subdividing**:
{• *effector abstract sc-agent*
• *receptor abstract sc-agent*
• *abstract software sc-agent implemented in the SCP Language*
}

atomic abstract sc-agent
⇒ *subdividing**:
{• *platform-independent abstract sc-agent*
• *platform-dependent abstract sc-agent*
}

Platform-independent abstract sc-agents include *atomic abstract sc-agents* implemented in the basic programming language of the OSTIS Technology, i.e. in the *SCP Language*.

When describing ***platform-independent abstract sc-agents***, platform independence is understood as platform independence from the point of view of the OSTIS Technology, i.e. implementation in a specialized programming language focused on processing semantic networks (*SCP Language*), since *atomic sc-agents* implemented in the specified language can be freely transferred from one ostis-platform to another. At the same time, programming languages that are traditionally considered platform-independent in this case cannot be considered as such.

There are *sc-agents* that fundamentally cannot be implemented at a platform-independent level, for example, the actual *sc-agents* for interpreting *sc-models* or receptor

and effector *sc-agents* that provide interaction with the external environment.

Platform-dependent abstract sc-agents include *atomic abstract sc-agents* implemented below the level of sc-models, i.e. not in the *SCP Language* but in some other program description language.

Each ***internal abstract sc-agent*** denotes a class of *sc-agents* that react to events in *sc-memory* and perform transformations exclusively within the same *sc-memory*.

Each ***effector abstract sc-agent*** denotes a class of *sc-agents* that react to events in *sc-memory* and perform transformations in an environment external to this *ostis-system*.

Each ***receptor abstract sc-agent*** designates a class of *sc-agents* that react to events in the environment external to this *ostis-system* and perform transformations in the memory of this system.

Each ***abstract sc-agent that is not implemented in the SCP Language*** must be implemented at the level of the *ostis-platform*, including hardware one. Such *abstract sc-agents* include abstract sc-agents for interpreting scp-programs, as well as effector and receptor abstract sc-agents.

Each ***abstract sc-agent implemented in the SCP Language*** can be implemented in the *SCP Language*, that is, at the platform-independent level, but, if necessary, it can also be implemented at the platform level, for example, in order to improve performance.

Abstract sc-agents for interpreting scp-programs include *abstract sc-agents* that are not implemented at the platform-independent level, providing interpretation of *scp-programs* and *scp-meta-programs*, including the creation of *scp-processes*, the actual interpretation of *scp-operators*, as well as other auxiliary actions. In fact, agents of this class ensure the operation of sc-agents of higher levels (software sc-agents and sc-meta-agents) implemented in the SCP Language, in particular, ensure that these agents comply with the general principles of synchronization.

Abstract software sc-agents includes all *abstract sc-agents* that provide the basic functionality of the system, that is, its ability to solve certain problems. Agents of this class should work in accordance with the general principles of synchronizing the activities of subjects in sc-memory.

The purpose of ***abstract sc-meta-agents*** is to coordinate the activities of *abstract software sc-agents*, in particular, solving the problem of interlocks. Agents of this class can be implemented in the SCP Language, however, other principles are used to synchronize their activities, respectively, to implement such agents, a different level of the SCP Language is required, the typology of which operators is completely similar to the typology of scp-operators, however, these operators have different operational semantics, taking into account differences in the principles of synchronization (working with *locks**). Programs of such a language will be called *scp-meta-programs*, corresponding to them *processes in sc-memory – scp-meta-processes*, operators – *scp-meta-operators*.

decomposition of an abstract sc-agent*
∈ *decomposition relation*

The ***decomposition of an abstract sc-agent**** relation interprets *non-atomic abstract sc-agents* as collectives of simpler *abstract sc-agents* interacting through *sc-memory*.

In other words, ***decomposition of an abstract sc-agent**** into *abstract sc-agents* of a lower level clarifies one of the possible approaches to the implementation of this *abstract sc-agent* by building a collective of simpler *abstract sc-agents*.

sc-agent
:= [agent on sc-memory]
⊂ *subject*
⇒ *family of subsets**:
abstract sc-agent

An ***sc-agent*** is understood as a concrete instance (from a set-theoretic point of view, an element) of some *atomic abstract sc-agent* operating in any particular intelligent system.

Thus, each *sc-agent* is a subject capable of performing some class of similar actions either only on *sc-memory* or on sc-memory and the external environment (for effector*sc-agents*). Each such action is initiated either by a state or situation in sc-memory, or by a state or situation in the external environment (for receptor sc-agents-sensors) corresponding to the initiation condition of the *atomic abstract sc-agent*, which instance is the specified *sc-agent*. In this case, an analogy can be drawn between the principles of object-oriented programming, considering an *atomic abstract sc-agent* as a class, and a specific *sc-agent* as an instance, a specific implementation of this class.

Interaction of *sc-agents* is carried out only through *sc-memory*. As a consequence, the result of the operation of any *sc-agent* is some change in the state of *sc-memory*, i.e. the deletion or generation of any *sc-elements*.

In general, one *sc-agent* can explicitly transfer control to another *sc-agent* if this *sc-agent* is known a priori. To do this, each *sc-agent* in *sc-memory* has an *sc-node* denoting it, with which it is possible to associate a specific situation in the current state of the knowledge base that the initiated *sc-agent* must process.

However, it is not always easy to determine the *sc-agent* which should take control from a given *sc-agent*, and therefore the situation described above occurs extremely rarely. Moreover, sometimes the condition for initiating

the *sc-agent* is the result of the activity of an unpredictable group of *sc-agents*, as well as the same construction can be the condition for initiating an entire group of *sc-agents*.

At the same time, not *sc-agent programs** communicate through *sc-memory* but the *sc-agents* themselves described by these programs.

In the process of work, the *sc-agent* can generate auxiliary *sc-elements* for itself, which it deletes after completing the act of its activity (these are auxiliary *structures* that are used as "information scaffolds" only during the execution of the corresponding act of activity and are deleted after the performance of the act).

sc-agent

⊃ *active sc-agent*

⇒ *first domain**:

- *key sc-elements of the sc-agent**
- *sc-agent program**
- *primary initiation condition**
- *initiation condition and result**

An ***active sc-agent*** is understood as an *sc-agent* of the ostis-system, which reacts to events corresponding to its initiation condition and, as a consequence, its *primary initiation condition**. The *sc-agents* that are not included in the set of ***active sc-agents*** do not respond to any events in *sc-memory*.

The connectives of the ***key sc-elements of the sc-agent**** relation link together the *sc-node*, denoting an *abstract sc-agent*, and the *sc-node*, denoting the set of *sc-elements*, which are key for a given *abstract sc-agent*, that is, given *sc-elements* are explicitly mentioned within programs implementing this *abstract sc-agent*.

The connectives of the ***sc-agent program**** relation link together the *sc-node*, denoting an *atomic abstract sc-agent*, and the *sc-node*, denoting a set of programs implementing the specified *atomic abstract sc-agent*. In the case of *platform-independent abstract sc-agent*, each connective of the *sc-agent program** relation connects the *sc-node* denoting the specified *abstract sc-agent* with a set of *scp-programs* describing the activities of this *abstract sc-agent*. This set contains one *agent scp-program* and a random number (maybe none) of *scp-programs* that are necessary to execute the specified *agent scp-program*.

In the case of the *platform-dependent abstract sc-agent*, each connective of the *sc-agent* program* relation links the *sc-node* denoting the specified *abstract sc-agent* with a set of files containing the source texts of the program in some external programming language that implements the activity of this *abstract sc-agent*.

The connectives of the ***primary initiation condition**** relation link together the *sc-node*, denoting an *abstract sc-agent*, and a binary oriented pair describing the primary initiation condition of this *abstract sc-agent*, i.e. such a specification of the *situations* in *sc-memory*, the occurrence of which prompts the *sc-agent* to switch to the active state and start checking for its full initiation condition.

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

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

After an event occurs in *sc-memory*, all *active sc-agents* are activated, the ***primary initiation condition**** of which corresponds to the event that occurred.

The connectives of the ***initiation condition and result**** relation link together the *sc-node*, denoting an *abstract sc-agent*, and a binary oriented pair linking the initiation condition for this *abstract sc-agent* and the results of executing this instance of the given *sc-agent* in any particular system.

The specified oriented pair can be considered as a logical implication connective, while the universality quantifier is implicitly imposed on *sc-variables* present in both parts of the connective and the existence quantifier is implicitly imposed on *sc-variables* present either only in the premise or only in the conclusion.

The first component of the specified oriented pair is a logical formula describing the initiation condition for the described *abstract sc-agent*, that is, a construction whose presence in *sc-memory* prompts the *sc-agent* to begin work on changing the state of *sc-memory*. This logical formula can be both atomic and non-atomic, in which the usage of any logical language connectives is allowed.

The second component of the specified oriented pair is a logical formula describing the possible results of the execution of the described abstract *sc-agent*, that is, a description of the changes in the state of *sc-memory* made by it. This logical formula can be both atomic and non-atomic, in which the usage of any logical language connective is allowed.

description of the behavior of an sc-agent

⊂ *semantic neighborhood*

The ***description of the behavior of an sc-agent*** is a *semantic neighborhood* describing the activity of an *sc-agent* to some degree of detail, however, such a description must be strict, complete, and unambiguously understood. Like any other *semantic neighborhood*, the ***description of the behavior of an sc-agent*** can be translated into any understandable, generally accepted

means that do not require special study, for example, into natural language.

The described *abstract sc-agent* is included in the corresponding ***description of the behavior of an sc-agent*** under the *key sc-element′* attribute.

VIII. Principles of synchronizing the activities of sc-agents

A. Clarification of the typology of processes in sc-memory, concepts of locks and locks classification

The concepts of an *action in sc-memory* and a *process in sc-memory* (information process performed by an agent in semantic memory) are synonymous, since all processes occurring in sc-memory are conscious and are performed by some sc-agents. Nevertheless, when it comes to synchronizing the execution of any transformations in the memory of a computer system, it is accepted in the literature to use the terms "process" and "interaction of processes" [15], [16], in connection with which we will use this term when describing the principles of synchronizing the activities of sc-agents when they perform parallel processes in sc-memory.

process in sc-memory
⇒ *subdividing**:
  {• *process in sc-memory corresponding to a platform-dependent sc-agent*
   • *scp-process*
     ⇒ *subdividing**:
       {• *scp-process that is not an scp-meta-process*
        • *scp-meta-process*
       }
  }

process in sc-memory corresponding to a platform-dependent sc-agent
⇒ *subdividing**:
  {• *process in sc-memory that corresponds to a platform-dependent sc-agent and is not an action of an abstract scp-machine*
   • *action of an abstract scp-machine*
     ⊃ *action of scp-program interpretation*
  }

To synchronize the execution of *processes in sc-memory*, it is proposed to use a locking mechanism based on existing algorithms for synchronizing information processes in traditional systems [15], [16]. As a possible direction for the development of this approach, it is possible to indicate the ideas of lock-free algorithms that are gaining popularity [17].

The ***lock**** relation connects the signs of *actions in sc-memory* with the signs of *structures* (situational ones) that contain elements that are blocked for the duration of performing this action or for some part of this period. Each such *structure* belongs to one of the *lock types*.

The first component of the connective of the ***lock**** relation is the sign of an *action in sc-memory*, the second is the sign of the blocked *structure*.

lock*
∈ *binary relation*

lock type
∋ *full lock*
∋ *lock on any change*
∋ *lock on deletion*

The ***lock type*** set contains all possible lock classes, i.e. *structures* containing *sc-elements* blocked by some *sc-agent* for the duration of performing some *action in sc-memory*.

Each *structure* belonging to the ***full lock*** set contains *sc-elements*, viewing and modification (deletion, addition of incident *sc-connectors*, deletion of the *sc-elements* themselves, changing the contents in the case of a file) which are prohibited to all *sc-agents*, except for the *sc-agent* itself, which performs the corresponding *action in sc-memory* associated with it by the *lock** relation.

In order to exclude the possibility of implementing *sc-agents*, which can make changes to the constructions describing the locks of other *sc-agents*, all elements of these constructions, including the sign of the *structure* containing the blocked *sc-elements* (belonging to both the ***full lock*** set and any other *lock type*) and the connectives of the *lock** relation linking this *structure* and a specific *action in sc-memory* are added to the ***full lock***, corresponding to the given *action in sc-memory*. Thus, each ***full lock*** corresponds to an affiliation loop linking its sign to itself.

Each *structure* belonging to the ***lock on any change*** set contains *sc-elements*, modification (physical deletion, addition of incident *sc-connectors*, physical deletion of *sc-elements*, changing the contents in the case of a file), which is prohibited to all *sc-agents*, except for the *sc-agent* itself, which performs the corresponding *action in sc-memory* associated with it by the *lock** relation. However, viewing (reading) of these *sc-elements* by any *sc-agent* is not prohibited.

Each *structure* belonging to the ***lock on deletion*** set contains *sc-elements*, the deletion of which is prohibited to all *sc-agents*, except for the *sc-agent*, which performs an action corresponding to this structure *in sc-memory*, associated with it by the *lock** relation. However, it is not prohibited to view (read) these *sc-elements* by any *sc-agent*, adding incident sc-connectors.

B. Principles of working with locks

Let us consider the principles of working with locks:

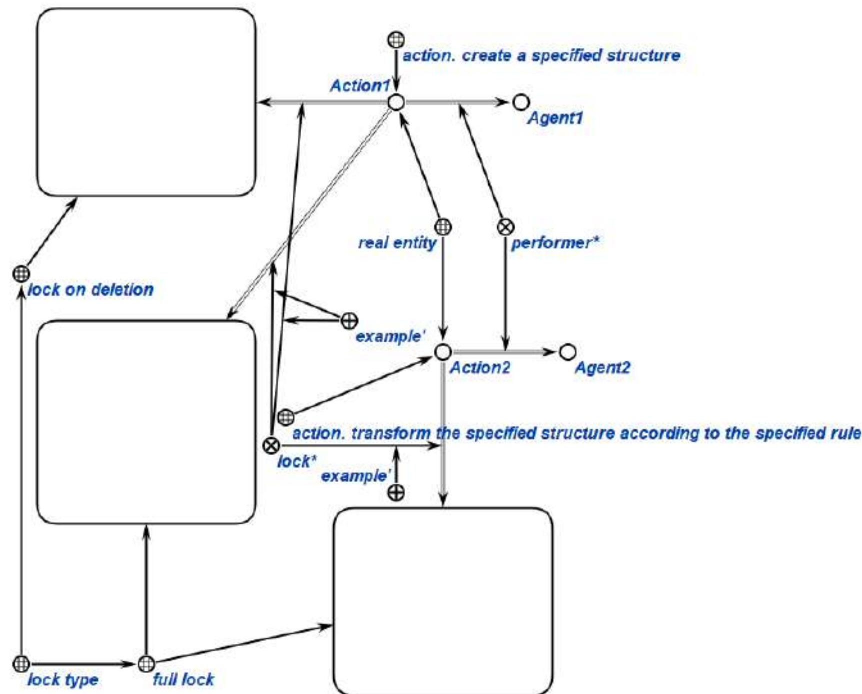

Figure 1. An example of using locks

- at any given moment, only one lock of each type can correspond to one process in sc-memory;
- at any given time, only one lock can correspond to one process in sc-memory, set on some specific sc-element;
- at the end of any process execution in sc-memory, all the locks set by it are automatically deleted;
- to increase the efficiency of the system as a whole, each process must block the minimum required set of sc-elements at any given time, removing the lock from each sc-element as soon as it becomes possible (safe);
- In the case when more particular subprocesses are explicitly allocated within the *process in sc-memory* (using the *temporal part*, sub-action*, action decomposition*, etc. relations*), then each such subprocess from the point of view of synchronizing execution is considered as an independent process, which can correspond to all necessary locks.
  - all child processes in sc-memory have access to the locks of the maternal process in the same way as if they were locks corresponding to each of such child processes;
  - in turn, the maternal process does not have any privileged access to sc-elements blocked by child processes and works with them in the same way as any other process in sc-memory. The exception is sc-elements denoting the child processes themselves, since the maternal process must be able to control the child one, for example, suspending or terminating their execution;
  - all child processes in relation to each other work the same way as in relation to any other processes;
  - in the case when the maternal process suspends execution (becomes a *deferred action*), all of its child processes also suspend execution. In turn, suspending one of the child processes in general does not explicitly initiate the stopping of the entire maternal process and, accordingly, other child processes.

Let us consider the principles of working with *full locks*:

- if the sc-element incident to some sc-connector gets into any full lock, then this sc-connector itself is also considered blocked by the same lock by default. The contrary is generally not true, since part of the sc-connectors incident to some sc-element may be completely blocked, while this element itself will not be blocked. This situation is typical, for example, for sc-nodes denoting classes of concepts;
- each process in sc-memory can freely modify or delete any sc-elements that get into the full lock corresponding to this process.

The principles of working with *full locks*, on the one hand, are the simplest, since all processes, except for the one who set such a lock, do not have access to the blocked sc-elements, and conflicts cannot arise. On the other hand, the frequent usage of locks of this type can

lead to the case when the system will not be able to fully use its knowledge and give incomplete or even incorrect answers to the questions posed.

Let us consider the principles of working with *locks on any change* and *locks on deletion*:

- only one lock of the same type can be set on the same sc-element at one time, but different processes can simultaneously set two different locks types on the same element. This concerns the case when the first process has set a lock on deletion on some sc-element and the second process then sets a lock on any change. In other cases, a lock conflict occurs;
- setting a lock of any type is also considered a change, so if a lock on any change was set on some sc-element, then another process will not be able to set a lock of any type on the same sc-element until the first process deletes its own;
- if a lock on deletion is set on some sc-connector, then by default the same lock is set on sc-elements that are incident to this sc-connector, since deleting these elements will lead to the deletion of this connector.

process in sc-memory
:= [action in sc-memory]
⇒ *subdividing**:
Classification of processes in sc-memory in terms of synchronizing their execution
= {• *action of searching for sc-elements*
• *action of generating sc-elements*
• *action of deleting sc-elements*
• *action of setting a lock of some type on some sc-element*
• *action of removing the lock from some sc-element*
}

In some cases, in order to ensure synchronization, it is necessary to combine several elementary actions on sc-memory into one indivisible action (***transaction in sc-memory***), for which it is guaranteed that no third-party process will be able to read or modify the sc-elements involved in this action, until the action completes. At the same time, unlike a situation with a full lock, a process, trying to access such elements, does not continue execution as if these elements simply did not exist in sc-memory but waits for the transaction to complete, after which it can perform any actions with these elements according to the general principles of process synchronization. The problem of ensuring transactions cannot be solved at the SC-code level and requires the implementation of such indivisible actions at the level of the *ostis-platform*.

If an *action of searching for sc-elements* is performed, all sc-elements found and saved within any process get into the corresponding *lock on any change* for this process. Thus, the integrity of the fragment of the knowledge base with which some process is working in sc-memory is guaranteed. In this case, the search and automatic setting of such a lock should be implemented as a *transaction in sc-memory*.

This approach also allows avoiding a situation where one process has blocked some sc-element on any change, and the second process is trying to generate or delete an *sc-connector* incident to this *sc-element*. In this case, the second process will have to first find and lock the specified *sc-element* on any change, which will cause a lock conflict (*interlock**).

In the case of generation of any sc-element within a certain process, it automatically gets into a full lock corresponding to this process. At the same time, the generation and automatic setting of such a lock should be implemented as a *transaction in sc-memory*. If necessary, the generated elements can be deleted (i.e. their temporary existence will not affect the activities of other processes at all) or unblocked when information is generated that may have some value in the future.

If any process tries to set a lock of any type on any sc-element already blocked by some other process, then, on the one hand, the lock cannot be set until another process unlocks the specified sc-element; on the other hand, in order to provide the possibility of searching and eliminating *interlocks*, it is necessary to explicitly indicate the fact that some process wants to access some sc-element blocked by another process. In order to be able to specify which processes are trying to block an already blocked *sc-element*, it is proposed, along with the *lock** relation, to use the *planned lock** relation, completely analogous to the *lock** relation.

The described mechanism also regulates the search processes, since the searching and saving of some sc-element involves the setting of a *lock on any change*. In addition, it should be taken into account that a *lock on any change* can be set on one sc-element after the *lock on deletion* corresponding to another process. In this case, there is no need to use the *planned locks** relation.

The action of checking for the presence of a lock on some sc-element and, depending on the result of the check, the setting of the lock or the planned lock (indicating the priority, if necessary) should be implemented as a transaction.

planned lock*
⊂ *lock**

The process to which the *planned lock** is assigned suspends execution until the already set locks are removed, after which the *planned lock** becomes a real *lock**, and the process continues execution in accordance with the general rules.

lock priority*
⇒ *scope of definition**:
*planned lock**

In the case when several processes are planning to set a lock on the same sc-element at once, the *lock priority** relation is used, linking the *planned lock** relation pairs. As a rule, the lock priority is determined by which of the processes previously tried to set a lock on the given sc-element, although in general the priority can be set or changed depending on additional criteria.

In the case of an attempt to delete some sc-element by some process, deletion can be carried out only if no lock is set (and is not planned to be set) on this sc-element by any other process.

In other cases, it is necessary to ensure that all processes working with this sc-element are completed correctly, and only then delete it physically.

To implement this possibility, each process can be matched with a set of sc-elements that are deleted by this process.

The action of checking for locks or planned locks on the deleted sc-element and actually deleting it or adding it to the set of deleted sc-elements for the corresponding process should be implemented as a transaction.

deleted sc-elements*
⇒ *first domain**:
process in sc-memory

Sc-elements that have got into the set of deleted sc-elements of some process in sc-memory are available to processes that have already set (or plan to set) locks on these sc-elements earlier (before attempting to delete it), and for all other processes these sc-elements are already considered deleted. A process trying to delete an sc-element suspends its execution until all processes, which have blocked and plan to block this sc-element, unlock it. In general, one sc-element can be included in the sets of deleted elements simultaneously for several processes, in this case, all such processes will simultaneously continue execution after removing all locks from this sc-element. If the deletion is attempted by one of the processes that has already set a lock on the specified sc-element, then the algorithm of actions remains the same – the sc-element is added to the set of sc-elements being deleted by this process and will be physically deleted as soon as all other processes that have set a lock on this sc-element remove them.

Let us consider the algorithm for removing the lock from some sc-element:

1) if one or more *planned locks** are set on this sc-element, then the first of them by priority (or the only one) becomes a *lock**, the corresponding process continues execution (becomes a real entity); the connective of the execution priority relation corresponding to the remote connective of the *planned lock** relation is also deleted, i.e. the priority is shifted by one position;
2) if there are no *planned locks** set on this sc-element, but it gets into the set of deleted sc-elements for one or more processes, then the given sc-element is physically deleted and the processes, suspended before its deletion, continue their execution (become real entities);
3) if the planned locks are not set on this sc-element and it is not included in the set of deleted ones for any process, then the lock is simply removed without any additional changes.

transaction in sc-memory
⇒ *subdividing**:
{
- *searching for some construction in sc-memory and automatic setting a lock on any change to the found sc-elements*
- *generating some sc-element and automatic setting of a full lock on it*
- *checking for the presence of a lock on some sc-element and, depending on the result of the check, setting a lock or a planned lock*
- *checking for the presence of locks or planned locks on the deleted sc-element and actually deleting it or adding it to the set of deleted sc-elements for the corresponding process*
- *removing the lock from a given sc-element and, if necessary, setting the first in priority planned lock or deleting this sc-element if it is included in the set of deleted sc-elements for some process*
- *searching for subprocesses of a process and adding them to a set of deferred actions in the case of adding the process itself to this set*
- *searching for subprocesses of a process and deleting them from the set of deferred actions if the process itself is deleted from this set*

}

C. Principles of synchronizing sc-agents implemented at the platform-independent level

When implementing *abstract software sc-agents* in the *SCP Language*, compliance with all the principles of synchronization of processes corresponding to these sc-agents is ensured at the level of *sc-agents for interpreting scp-programs*, i.e. by means of the *ostis-platform*. When implementing *abstract software sc-agents* at the platform level, compliance with all synchronization principles is assigned, firstly, directly to the agent developer and,

secondly, to the platform developer. For example, the platform can provide access to elements stored in sc-memory through some programming interface that already takes into account the principles of working with locks, which will save the agent developer from having to take into account all these principles manually.

In addition, a number of specific principles of operation of *abstract software sc-agents*, implemented in the *SCP Language*, are highlighted:

- as a result of the appearance in sc-memory of some construction that satisfies the condition of initiating some *abstract sc-agent* implemented using the *SCP Language*, an *scp-process* is generated and initiated in *sc-memory*. As a template for generation, an *agent scp-program* is used, corresponding to this *abstract sc-agent*.
- each such *scp-process* corresponding to some *agent scp-program* can be associated with a set of structures describing locks of various types. Thus, synchronization of interaction of parallel *scp-processes* is carried out in the same way as in the case of any other *actions in sc-memory*.
- despite the fact that each *scp-operator* is an atomic action in sc-memory, which is a sub-action within the entire *scp-process*, locks corresponding to one operator are not introduced to avoid the lengthiness and excess of additional system constructions created when executing some *scp-process*. Instead of it, locks that are common to the entire *scp-process* are used. Thus, *agents for interpreting scp-programs* work only taking into account the locks common to the entire interpreted *scp-process*.
- processes describing the activity of agents for interpreting *scp-programs* are usually not created, therefore, their corresponding locks are not introduced. Since such agents work with a unique scp-process and their number is limited and known, then the usage of locks for their synchronization is not required.
- if the *scp-process* is suspended (is added to the set of *deferred actions*), in accordance with the general synchronization rules, all its child processes must also be suspended. In this regard, all *scp-operators*, which at this moment are *real entities*, become *deferred actions*.
- in order to avoid undesirable changes in the body of the *scp-process*, the entire construction generated on the basis of some *scp-program* (the entire *sc-text* describing the decomposition of the *scp-process* into *scp-operators*) must be added to the *full lock* corresponding to this *scp-process*.
- if necessary, the corresponding *scp-operators* of the *scp-operator for lock control* class are used to unlock or lock some construction by some lock type.
- after completing the execution of some scp-process, its text is usually deleted from *sc-memory* and all blocked constructions are released (signs of structures that denoted locks are destroyed).
- as a rule, the particular *action class* corresponding to a specific *scp-program* is not explicitly introduced, but the more general *scp-process* class is used, except in cases when the introduction of a special *action class* is necessary for some other reasons.

In general, the entire locking mechanism can be described both at the SC-code level (to increase the level of platform independence) and, if necessary, can be implemented at the *ostis-platform* level, for example, to improve performance. To do this, a unique table, containing a list of blocked elements with an indication of the lock type at each time, can be assigned to each process executed in sc-memory at the lower level.

D. Example of the operation of the locking mechanism

Let us consider an example of using the described mechanism.

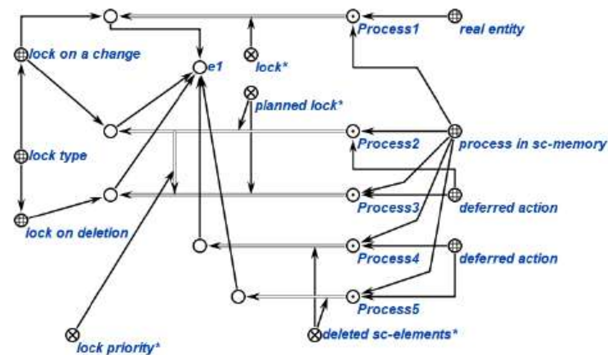

Figure 2. An example of using planned locks

In this example, *Process1* works directly with the sc-element ***e1***,*Process2* and *Process3* plan to set a lock on any change and a lock on deletion, respectively, at the same time, *Process2* tried to set its lock before *Process3*, therefore, according to the direction of the connective of the *lock priority** relation, its lock will be set earlier. *Process4* and *Process5* are waiting for all locks and planned locks to be removed, after which ***e1*** will be deleted, and *Process1* and *Process2* will continue their execution. No other planned locks can be set anymore, since ***e1*** got into a set of deleted sc-elements of at least one process and, in accordance with the rules set out above, all other processes except *Process1–Process5* can no longer access this sc-element. The executed process belongs to the real entity set, suspended – to the deferred action set.

After *Process1* has unlocked sc-element ***e1***, this element will be locked by *Process2*, and *Process2* will continue execution. *Planned lock** set by *Process2*, becomes a regular *lock**.

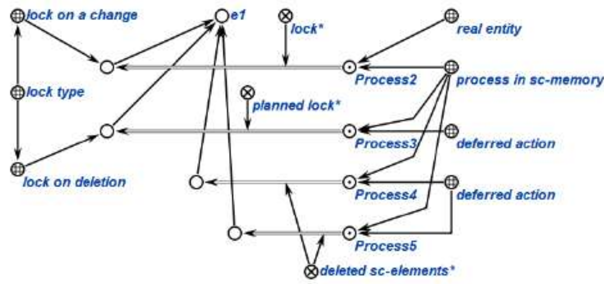

Figure 3. An example of using planned locks (continued)

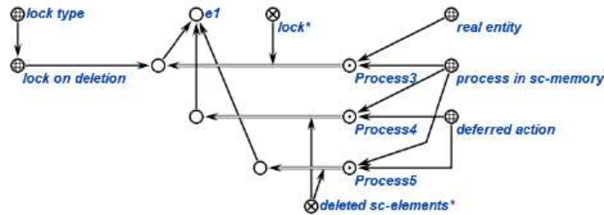

Figure 4. An example of using a lock on deletion

After *Process2* has unlocked sc-element ***e1***, this element will be locked by *Process3*, and *Process3* will continue execution.

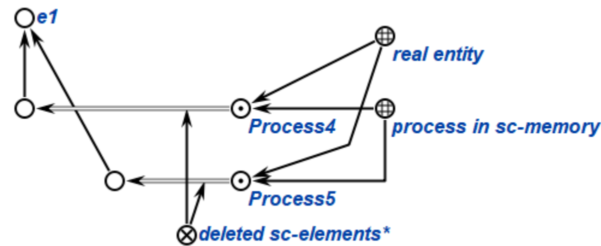

Figure 5. The sc-elements to be deleted

When all processes remove the locks from sc-element ***e1***, it can be physically deleted, and *Process4* and *Process5* will continue execution.

Depending on the specific *lock types* set by parallel processes on some sc-elements and what specific actions with these *sc-elements* are supposed to be performed further within these processes, the interlock situations are possible when each of these processes will wait for the second process to remove the lock from the desired *sc-element*, without removing the lock set by itself from the *sc-element*, access to which is required by the second process.

In the case when at least one of the locks is a *full lock*, an interlock situation cannot occur, since *sc-elements* that have got into the *full lock* of some *scp-process* are not available to other *scp-processes*, even for reading, and, thus, the rest of the *scp-processes* will work as if the blocked *sc-elements* are simply missing in the current state of *sc-memory*.

In cases where none of the set locks is a *full lock*, interlocks may occur.

Elimination of the *interlock* is impossible without the intervention of a specialized *sc-meta-agent*, which has the right to ignore the locks set by other processes.

In general, the problem of a specific interlock can be solved by performing the following steps by a specialized *sc-meta-agent*:

- rollback of several operations performed by one of the processes involved in the interlock by as many steps back as necessary so that the second process gets access to the necessary *sc-elements* and can continue execution;
- waiting for the execution of the second process until it completes or removes all locks from *sc-elements* that the first process needs to access;
- repeated execution of canceled operations within the first process and continuation of its execution but taking into account the changes in memory made by the second process.

For *sc-meta-agents*, all sc-elements, including those describing locks, planned locks, etc., are completely equivalent to each other in terms of access to them, i.e. any *sc-meta-agent* has access to any sc-elements, even those that have got into a full lock for any other process. This is necessary so that *sc-meta-agents* can identify and fix various problems, for example, the interlock problem described above.

Thus, the problem of synchronizing the activities of *sc-meta-agents* requires the introduction of additional rules.

We will divide this problem into two more specific ones:

- ensuring synchronization of the activities of *sc-meta-agents* among themselves;
- ensuring synchronization of the activities of *sc-meta-agents* and *software sc-agents*.

The first problem is proposed to be solved by prohibiting parallel execution of *sc-meta-agents*. Thus, at any given time within one *ostis-system*, there can be only one process corresponding to the *sc-meta-agent* and being the *real entity*.

The second problem is proposed to be solved by introducing additional privileges for *sc-meta agents* when accessing any sc-element. One rule is enough for this:

If a certain sc-element has become used within a process corresponding to the *sc-meta-agent* (for example, it has become an element of at least one scp-operator included in this process), then all processes, into the locks corresponding to which the specified sc-element gets, become deferred actions (suspend execution). As soon as the specified sc-element ceases to be used within the process corresponding to the *sc-meta-agent*, all processes suspended for this reason continue execution.

The considered limitations do not significantly deteriorate the performance of the ostis-system, since *sc-meta-agents* are designed to solve a fairly narrow class of problems, which, as the experience of practical developing prototypes of various *ostis-systems* has shown, arise quite rarely.

It is worth noting that there may be a situation in which the execution of some process in sc-memory is interrupted due to an error. In this case, there is a possibility that the lock set by this process will not be removed until the sc-meta-agent that has detected a similar situation does that. However, this problem can only be partially solved at the sc-model level, for cases when an error occurs during the interpretation of the scp-program, is tracked by the scp-interpreter, and a corresponding construction is formed in memory that reports the problem to the sc-meta-agent. Cases where an error has occurred at the scp-interpreter or sc-storage levels should be considered at the ostis-platform level.

IX. Basic programming language of ostis-systems

The allocation of the Basic programming language for ostis-systems allows for a clear separation of the level of methods and, accordingly, the skills of the ostis-system, which can be fully described at the level of the knowledge base, and lower-level skills that provide interpretation of these higher-level skills. In other words, the allocation of such a language allows for the platform independence of ostis-systems, both in the case of a software implementation of the ostis-platform and in the case of an *associative semantic computer*.

As a basic language for describing programs for processing texts of the *SC-code*, the *SCP Language* is proposed.

The *SCP Language* is a graph procedural programming language designed for efficient processing of *sc-texts*. The *SCP Language* is a parallel asynchronous programming language.

SCP Language
:= *frequently used sc-identifier**:
[scp-program]

The data representation language for texts of the *SCP Language* (*scp-programs*) is the *SC-code* and, accordingly, any variants of its external representation. The *SCP Language* is built on the basis of the *SC-code*, as a result of which *scp-programs* can be part of the processed data for *scp-programs*, including in relation to themselves. Thus, the *SCP Language* provides the ability to build reconfigurable programs. However, in order to be able to reconfigure the program directly in the process of its interpretation, it is necessary at the level of the interpreter of the *SCP Language (Abstract scp-machine)* ensure the uniqueness of each executable copy of the source program. Such an executable copy generated on the basis of the *scp-program* will be called an *scp-process*. The inclusion of the sign of some *action in sc-memory* in the set of *scp-processes* guarantees the fact that only the signs of elementary actions (*scp-operators*) will be present in the decomposition of this action, which can be interpreted by the implementation of the *Abstract scp-machine* (interpreter of scp-programs).

The *SCP Language* is considered as an assembler for an *associative semantic computer* [3].

Abstract scp-machine
∈ *scp-machine*
⇐ *generalized model**:
scp-interpreter

The *basic model for processing sc-texts* includes the *Subject domain of the Basic programming language of ostis-systems*, that is, a description of the syntax and denotational semantics of the SCP Language, as well as a description of the *Abstract scp-machine* that is a model of the *scp-interpreter*, which should be part of the *ostis-platform* (although in general there can exist platform variants that do not contain such an interpreter, which, however, will not allow using the advantages of the proposed basic model).

Let us consider the key features and advantages of the *Basic model for processing sc-texts*:

- The texts of the *SCP Language* programs are written using the same unified semantic networks as the processed information, so we can say that the *Syntax of the SCP Language* at the basic level is the same as the *Syntax of the SC-code*.
- An approach to interpreting *scp-programs* involves creating a unique *scp-process* at each call of the *scp-program*.
- Several independent *sc-agents* can be executed simultaneously in shared memory, while different copies of *sc-agents* can be executed on different servers, due to the distributed implementation of the ostis-platform. Moreover, the *SCP Language* allows making parallel asynchronous calls to subprograms with subsequent synchronization and even executing operators in parallel within a single *scp-program*.
- The transfer of the *sc-agent* from one system to another consists in a simple transfer of a fragment of the knowledge base, without any additional operations depending on the interpretation platform.
- The fact that the specifications of *sc-agents* and their programs can be written in the same language as the processed knowledge significantly reduces the list of specialized tools intended for designing knowledge processing machines and simplifies their development by using more universal components.

- The fact that a unique *scp-process* is created for the interpretation of the *scp-program* makes it possible to optimize the execution plan before its implementation and even directly during execution without the potential danger of ruining the general universal algorithm of the entire program. Moreover, such an approach to the design and interpretation of programs allows talking about the possibility of creating self-reconfigurable programs.

A. Concept of an scp-program

scp-program
⊂ *program in sc-memory*
⊃ *agent scp-program*

Each ***scp-program*** is a *generalized structure* describing one of the decomposition options for actions of some class performed in sc-memory. The sign of the *sc-variable* corresponding to a specific decomposable action is a *key sc-element′* within the ***scp-program***. It is also explicitly indicated that this sign belongs to the set of *scp-processes*.

Thus, each ***scp-program*** describes in a generalized form the decomposition of some *scp-process* into interrelated *scp-operators*, indicating, if any, arguments for this *scp-process*.

Agent scp-programs are a special case of *scp-programs* in general, however, they deserve separate consideration, since they are used most often. *Scp-programs* of this class are implementations of programs of knowledge processing agents and have a rigidly fixed set of parameters. Each such program has exactly two *in-parameters′*. The value of the first parameter is the sign of a binary oriented pair, which is the second component of the connective of the *primary initiation condition** relation for an abstract *sc-agent*, the set of *sc-agent programs** of which includes the considered ***agent scp-program***, and in fact, it describes a class of events, to which the specified sc-agent responds.

The value of the second parameter is an *sc-element*, which is directly associated with the event, as a result of which the corresponding *sc-agent* was initiated, i.e., for example, generated or deleted *sc-arc* or *sc-edge*.

Let us consider the principles of implementing *abstract sc-agents implemented in the SCP Language*:

- general principles of the organization of interaction between *sc-agents* and users of the *ostis-system* through a shared *sc-memory*;
- as a result of the appearance in sc-memory of some construction that satisfies the condition of initiating some *abstract sc-agent* implemented using the *SCP Language*, the *scp-process* is generated and initiated in *sc-memory*. As a template for generation, an *agent scp-program* is used, specified in the set of programs of the corresponding *abstract sc-agent*;
- each such *scp-process* corresponding to some *agent scp-program* can be associated with a set of structures describing locks of various types. Thus, synchronization of interaction of parallel *scp-processes* is carried out in the same way as in the case of any other *actions in sc-memory*;
- Within the *scp-process*, child *scp-processes* can be created, but synchronization between them, if necessary, is carried out by introducing additional internal locks. Thus, each *scp-process* from the point of view of *processes in sc-memory* is atomic and complete act of activity of some *sc-agent*;
- in order to avoid undesirable changes in the body of the *scp-process* itself, the entire structure generated on the basis of some *scp-program* (the entire text of the *scp-process*) should be added to the *full lock* corresponding to this *scp-process*;
- all constructions generated during the execution of the *scp-process* automatically get into the *full lock* corresponding to this *scp-process*. Additionally, it should be noted that the sign of this structure itself and all meta-information about it are also included in this structure;
- if necessary, it is possible to manually unlock or lock some construction with some lock type using the corresponding *scp-operators* of the *scp-operator for lock control* class;
- after completing the execution of some *scp-process*, its text is usually deleted from *sc-memory*, and all blocked constructions are released (signs of structures that denoted locks are destroyed).

B. Concept of an scp-process

An ***scp-process*** is understood as some *action in sc-memory* that uniquely describes a specific act of executing some *scp-program* for given source data. If the *scp-program* describes an algorithm for solving a problem in a general way, then the *scp-process* denotes a specific action that implements this algorithm for the specified input parameters.

In fact, the ***scp-process*** is a unique copy created on the basis of the *scp-program*, in which each *sc-variable*, with the exception of *scp-variables′*, corresponds to the generated *sc-constant*.

Belonging of some action to a set of *scp-processes* guarantees the fact that only signs of elementary actions (*scp-operators*) will be present in the decomposition of this action, which can be interpreted by the implementation of an *Abstract scp-machine*.

C. Concept of an scp-operator

Each ***scp-operator*** represents some elementary *action in sc-memory*. The arguments of the *scp-operator* will be called operands. The order of the operands is specified using the appropriate role relations (*1′*, *2′*, *3′*, and so on). The operand marked with role relation *1′* will be called the first operand, marked with role relation *2′* – the second operand, etc. The type and meaning of each

operand is also specified using various subclasses of the *scp-operand'* relation. In general, as the operand, any *sc-element* can act, including the sign of any *scp-program*, including the program itself containing this operator.

Each ***scp-operator*** must have one or more operands, as well as an indication of the ***scp-operator*** (or several) that should be executed next. The exception to this rule is the *scp-operator for program completion*, which does not contain a single operand and after which execution no *scp-operators* can be executed within this program.

Each ***atomic type of the scp-operator*** is a class of *scp-operators*, which is not divided into more particular ones and, accordingly, is interpreted by the implementation of the *Abstract scp-machine*.

Let us consider the upper level of the classification of scp-operators, which is given in more detail in [3].

scp-operator
⊂ *action in sc-memory*
⇐ *family of subsets**:
atomic type of the scp-operator
⇒ *subdividing**:
{
- *scp-operator for generating constructions*
- *scp-operator for associative search of constructions*
- *scp-operator for deleting structures*
- *scp-operator for checking conditions*
- *scp-operator for controlling the values of operands*
- *scp-operator for controlling scp-processes*
- *scp-operator for event control*
- *scp-operator for processing files contents*
- *scp-operator for lock control*

}

The role relation ***initial operator'*** specifies those *scp-operators* that should be executed first within the decomposition of the *scp-process* that corresponds to the *scp-program*, i.e. those with which, actually, the execution of the *scp-process* begins.

D. Parameters of scp-programs

parameters of the scp-program'
⊂ *action argument'*
⇒ *subdividing**:
{
- *in-parameter'*
- *out-parameter'*

}

The role relation ***parameter of the scp-program'*** links the sign of the *scp-process* with its arguments, that corresponds to the *scp-program*.

Parameters of the ***in-parameter'*** type, although they correspond to *variables of the scp-program'*, cannot change the value during its interpretation. The fixed value of the variable is set when creating a unique copy of the *scp-program* (*scp-process*) for its interpretation, and thus the corresponding *scp-variable'* at the time of its interpretation becomes an *scp-constant'* within each *scp-operator* in which this *scp-variable'* occurred. The usage of *in-parameters* can be considered by analogy with the usage of a variant of the value transfer mechanism in traditional programming languages, with the condition that the value of a local variable within a child program cannot be changed.

Parameters of the ***out-parameter'*** type correspond to *variables of the scp-program'* and have all the same corresponding properties. Most often, it is assumed that the value of this parameter is necessary for the maternal *scp-program* containing the call operator of the current *scp-program*. At the same time, at the moment of the beginning of interpretation, a node denoting a variable (or rather, its unique copy within the process) of the maternal process is passed directly to the child process as a parameter. The specified variable may, if necessary, have a value or not. After completion and during the interpretation of the child process, the maternal process can still work with the variable passed as the *out-parameter'*, viewing or changing its value if necessary. The usage of the out-parameter can be considered by analogy with the usage of the link transmission mechanism in traditional programming languages.

X. Model for the interpreter of the Basic programming language of ostis-systems

The advantages of the proposed multi-agent approach to building knowledge processing machines and, accordingly, problem solvers can work not only at the platform-independent level but also at lower levels. So, in particular, the interpreter of the *Basic programming language of ostis-systems* is also proposed to be built as a *non-atomic abstract sc-agent* that provides interpretation of the methods described in the *SCP Language*. Thus, such an interpreter is included in the general hierarchy of agents that build-up the *knowledge processing machine* of *ostis-systems* and is an *abstract sc-agent that is not implemented in the SCP Language*.

In general, there may be many options for implementing such interpreters. Within the *OSTIS Standard*, one of them is offered as a standard and is called an *Abstract scp-machine*.

Abstract scp-machine
∈ *abstract sc-agent that is not implemented in the SCP Language*
⇒ *decomposition of an abstract sc-agent**:
{
- *Abstract sc-agent for creating scp-processes*
- *Abstract sc-agent for interpreting scp-operators*

- *Abstract sc-agent for synchronizing the process of interpreting scp-programs*
- *Abstract sc-agent for destroying scp-processes*
- *Abstract sc-event for synchronizing events in sc-memory and its implementation*
  ⇒ *decomposition of an abstract sc-agent**:
    {• *Abstract sc-agent for translating the generated event specification in sc-memory into an internal representation*
    • *Abstract sc-agent for processing an event in sc-memory that initiates an agent scp-program*
    }

}

The purpose of an *Abstract sc-agent for creating scp-processes* is to create *scp-processes* corresponding to a given *scp-program*. This *sc-agent* is activated when an *initiated action* belonging to the *action of interpreting scp-program* class appears in *sc-memory*. After the *sc-agent* checks the initiation condition, the *scp-process* is created taking into account the specific parameters of the interpretation of the *scp-program*, after which the *initial operator'of the scp-process* is searched and added to the the set of *real entities*.

The purpose of the an *Abstract sc-agent for interpreting scp-operators* is actually the interpretation of the operators of the *scp-program*, that is, the execution in *sc-memory* of actions described by a specific *scp-operator*. This *sc-agent* is activated when an *scp-operator* belonging to the *real entities* class appears in *sc-memory*. After performing the action described by the *scp-operator*, the *scp-operator* is added to the set of *past entities*. In the case when the semantics of the action described by the *scp-operator* suggests the possibility of branching for the *scp-program* after executing this *scp-operator*, then one of the subsets of the class of *performed actions* – *unsuccessfully performed action* or *successfully performed action* is used.

The purpose of an *Abstract sc-agent for synchronizing the process of interpreting scp-programs* is to provide transitions between *scp-operators* within a single *scp-process*. This *sc-agent* is activated when some *scp-operator* is added to the set of *past entities*. Next, a transition is made along the *sc-arc* belonging to the *sequence of actions** relation (or more particular relations, if the *scp-operator* was added to the set of *successfully performed actions* or *unsuccessfully performed actions*). In this case, the next *scp-operator* becomes a *real entity* (active *scp-operator*) if at least one *scp-operator* associated with it by incoming *sc-arcs* belonging to the *sequence of actions** relation (or more particular relations) became a *past entity* (or, respectively, a subset of past entities). In the case when it is necessary to wait for the completion of all previous operators, the operator of the *conjunction of preceding operators* class is used for synchronization.

The purpose of an *Abstract sc-agent for destroying scp-processes* is the destruction of the *scp-process*, i.e. the deletion from *sc-memory* of all *sc-elements* that build it up. This *sc-agent* is activated when an *scp-process* belonging to a set of *past entities* appears in *sc-memory*. At the same time, the destroyed *scp-process* does not necessarily have to be fully formed. The need to destroy an incomplete *scp-process* may arise if, when creating the *scp-process*, problems arose that did not allow continuing the creation of the *scp-process* and its performance.

The purpose of an *Abstract sc-agent for event synchronization in sc-memory and its implementation* is to ensure the operation of *non-atomic sc-agents* implemented in the *SCP Language*.

The purpose of an ***Abstract sc-agent for translating the generated event specification in sc-memory into the internal representation*** is the translation of oriented pairs describing the *primary initiation condition** of some *sc-agent* into the internal representation of elementary events at the level of *sc-storage*, provided that this *sc-agent* is implemented at a platform-independent level (using the *SCP Language*). The condition for initiating this *sc-agent* is the appearance in *sc-memory* of a new element of the set of *active sc-agents*, for which the corresponding oriented pair will be found and translated.

The purpose of an *Abstract sc-agent for event processing in sc-memory, initiating the agent scp-program* is to search for an *agent scp-program*, included in the set of *sc-agent programs** for each *sc-agent*, the primary initiation condition of which corresponds to an event that occurred in *sc-memory*, as well as the generation and initiation of an action aimed at interpreting this program. As a result of the operation of this *sc-agent*, an *initiated action* appears in *sc-memory*, belonging to the *action* of *interpreting scp-program* class.

XI. Conclusion and directions for further development

In the article, the current problems in the field of developing hybrid problem solvers are formulated and an approach to solving some particular problems that are part of these more general problems is proposed. Thus, the solution of the formulated general problems is still relevant, however, the usage of the *OSTIS Technology* and the principles proposed in this work for constructing problem solvers based on it creates preconditions for their solution.

It is possible to formulate a number of more specific directions for the development of the approaches proposed in the article:

- Integrate ideas of situational control into the proposed approach more closely and fully;
- Refine the proposed locking mechanism, in particular, to minimize the number of lock classes, to take into account and implement the ideas of implementing lock-free algorithms;
- Eliminate the need to introduce sc-meta-agents and scp-meta-programs.
- Modify the *SCP Language* in order to be capable of describing the receptor and effector interaction of ostis-systems within scp-programs.
- When developing an Abstract scp-machine, to take into account the principles of building wave programming languages [18], [19] and the ideas of insertion programming and modeling [20], [21].

Acknowledgment

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

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

References

[1] A. Kolesnikov, *Gibridnye intellektual'nye sistemy: Teoriya i tekhnologiya razrabotki [Hybrid intelligent systems: theory and technology of development]*, A. M. Yashin, Ed. SPb.: Izd-vo SPbGTU, 2001.

[2] D. Shunkevich, "Agent-oriented models, method and tools of compatible problem solvers development for intelligent systems," in *Open semantic technologies for intelligent systems*, V. Golenkov, Ed. BSUIR, Minsk, 2018, pp. 119–132.

[3] 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.

[4] A. Narin'jani, "Ne-faktory: kratkoe vvedenie [non-factors: a brief introduction]," *Novosti iskusstvennogo intellekta [Artificial intelligence news]*, no. 2, pp. 52–63, 2004.

[5] V. Gorodetskii, V. Samoilov, and D. Trotskii, "Bazovaya ontologiya kollektivnogo povedeniya avtonomnykh agentov i ee rasshireniya [Basic ontology of autonomous agents collective behavior and its extension]," *Izvestiya RAN. Teoriya i sistemy upravleniya [Proceedings of the RAS. Theory and control systems]*, no. 5, pp. 102–121, 2015, (in Russian).

[6] M. Wooldridge, *An Introduction to MultiAgent Systems - Second Edition*. Wiley, 2009.

[7] V. Tarasov, *Ot mnogoagentnykh sistem k intellektual'nym organizatsiyam [From multi-agent systems to intelligent organizations]*. M.: Editorial URSS, 2002, (in Russian).

[8] L. Cao, "In-depth behavior understanding and use: The behavior informatics approach," *Information Sciences*, vol. 180, no. 17, pp. 3067–3085, Sep. 2010. [Online]. Available: https://doi.org/10.1016/j.ins.2010.03.025

[9] L. Cao, T. Joachims, C. Wang, E. Gaussier, J. Li, Y. Ou, D. Luo, R. Zafarani, H. Liu, G. Xu, Z. Wu, G. Pasi, Y. Zhang, X. Yang, H. Zha, E. Serra, and V. Subrahmanian, "Behavior informatics: A new perspective," *IEEE Intelligent Systems*, vol. 29, no. 4, pp. 62–80, Jul. 2014. [Online]. Available: https://doi.org/10.1109/mis.2014.60

[10] M. Pavel, H. B. Jimison, I. Korhonen, C. M. Gordon, and N. Saranummi, "Behavioral informatics and computational modeling in support of proactive health management and care," *IEEE Transactions on Biomedical Engineering*, vol. 62, no. 12, pp. 2763–2775, Dec. 2015. [Online]. Available: https://doi.org/10.1109/tbme.2015.2484286

[11] G. S. Al'tshuller, *Najti ideju: Vvedenie v TRIZ — teoriju reshenija izobretatel'skih zadach, 3-e izd. [Find an idea: An introduction to TRIZ - the theory of inventive problem solving, 3rd ed.]*. M.: Al'pina Pablisher, 2010.

[12] G. P. Shhedrovickij, *Shema mysledejatel'nosti – sistemno-strukturnoe stroenie, smysl i soderzhanie [Scheme of mental activity – system-structural structure, meaning and content]*. M.: Shk. kul't. pol., 1995.

[13] D. Pospelov, *Situacionnoe upravlenie. Teorija i praktika [Situational management. Theory and practice]*. M.: Nauka, 1986.

[14] D. Shunkevich, "Ontological approach to the development of hybrid problem solvers for intelligent computer systems," in *Open semantic technologies for intelligent systems*, V. Golenkov, Ed. BSUIR, Minsk, 2021, pp. 63–74.

[15] E. W. Dijkstra, *Cooperating Sequential Processes*. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 65–138.

[16] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 26, no. 1, p. 100–106, jan 1983. [Online]. Available: https://doi.org/10.1145/357980.358021

[17] B. Chatterjee, S. Peri, M. Sa, and K. Manogna, "Non-blocking dynamic unbounded graphs with worst-case amortized bounds." Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2022/15795/

[18] P. Sapaty, "Jazyk VOLNA-0 kak osnova navigacionnyh struktur dlja baz znanij na osnove semanticheskih setej [WAVE-0 language as a basis for navigational structures for knowledge bases based on semantic networks]," *Izv. AN SSSR. Tehn. kibernet. [Izv. Academy of Sciences of the USSR. Tech. cybernet.]*, no. 5, pp. 198–210, 1986.

[19] D. I. Moldovan and Y.-W. Tung, "SNAP: A VLSI architecture for artificial intelligence processing," *Journal of Parallel and Distributed Computing*, vol. 2, no. 2, pp. 109–131, 1985. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0743731585900310

[20] A. Letichevskij, J. Kapitonova, V. Volkov, V. Vyshemirskij, and A. Letichevskij (Jr.), "Insercionnoe programmirovanie [insertion programming]," *Kibernetika i sistemnyj analiz [Cybernetics and systems analysis]*, no. 1, pp. 19–32, 2003.

[21] A. Letichevskij, "Insercionnoe modelirovanie [insertion modeling]," *Upravljajushhie sistemy i mashiny [Control systems and machines]*, no. 6, pp. 3–14, 2012.

Гибридные решатели задач интеллектуальных компьютерных систем нового поколения

Шункевич Д.В.

В работе сформулированы актуальные проблемы текущего состояния технологий разработки гибридных решателей задач, предложен подход к их решению на основе Технологии OSTIS. Сформулированы принципы построения решателя задач как иерархической системы навыков, основанной на многоагентном подходе, приведены онтологии агентов и выполняемых ими действий. Сформулированы принципы синхронизации деятельности агентов, а также разработана онтология базового языка программирования для реализации программ агентов и модель интерпретатора такого языка.

Received 01.11.2022