

Current State of ostis-systems Component Design Automation Tools

Maksim Orlov, Anna Makarenko, Ksenija Petrochuk
*Belarusian State University of
Informatics and Radioelectronics*
Minsk, Belarus

Email: orlovmaksimkonst@gmail.com, anna.makarenko1517@gmail.com, xenija.petrotschuk@gmail.com

Abstract—In the article, an approach to the design of intelligent systems is considered, focused on the use of compatible reusable components, which significantly reduces the complexity of developing such systems. The key means of supporting the component design of intelligent computer systems is the manager of reusable components proposed in the work.

Keywords—Component design of intelligent computer systems; reusable semantically compatible components; knowledge-driven systems; semantic networks; OSTIS Technology.

I. Introduction

The main result of artificial intelligence is not the intelligent systems themselves, but powerful and effective technologies for their development. The analysis of modern technologies for designing intelligent computer systems shows that along with very impressive achievements, the following serious problems occur [1]–[3]:

- high requirements for the initial qualifications of users and developers. Artificial intelligence technologies are not focused on the wide range of developers and users of intelligent systems and, therefore, have not received mass distribution;
- modern information technologies are not oriented to a wide range of developers of applied computer systems;
- there is no general-unified solution to the problem of semantic compatibility of computer systems [4]. There are no approaches that allow integrating scientific and practical results in the field of artificial intelligence, which generates a high degree of duplication of results and a lot of non-unified formats for representation of data, models, methods, tools, and platforms;
- lack of powerful tools for designing intelligent computer systems, including intelligent training subsystems, subsystems for collective design of computer systems and their components, subsystems for verification and analysis of computer systems, subsystems for component design of computer systems;
- long terms of development of intelligent computer systems and high level of complexity of their maintenance and extension;

- the degree of dependence of artificial intelligence technologies on the platforms on which they are implemented is high, which leads to significant changes in technologies when transitioning to new platforms;
- the degree of dependence of artificial intelligence technologies on subject domains in which these technologies are used is high;
- there is a high degree of dependence of intelligent computer systems and their components on each other; the lack of their automatic synchronization. The absence of self-sufficiency of systems and components, their ability to operate separately from each other without loss of expediency of their use;
- increase in the time to solve the problem with the expansion of the functionality of the problem solver and with the expansion of the knowledge base of the system [5];
- lack of methods for designing intelligent computer systems. Updating computer systems often boils down to the development of various kinds of “patches”, which eliminate not causes of the identified disadvantages of updated computer systems but only some consequences of these causes;
- poor adaptability of modern computers to the effective implementation of even existing knowledge representation models and models for solving problems that are difficult to formalize, which requires the development of fundamentally new computers [6];
- there is no single approach to the allocation of reusable components and the formation of libraries of such components, which leads to a high complexity of reuse and integration of previously developed components in new computer systems.
- there is a variety of semantically equivalent implementations of problem-solving models, duplication of knowledge base and user interface components that differ not in the essence of these components but in the form of representation of the processed information;

To solve these problems, it is necessary to implement a

comprehensive technology for designing intelligent computer systems, which includes the following components:

- a model of an intelligent computer system [7];
- a *library of reusable components* and corresponding *tools to support component design of intelligent computer systems*;
- an intelligent integrated automation system for the collective design of intelligent computer systems, including subsystems for editing, debugging, performance evaluation, and visualization of developed components, as well as a simulation subsystem;
- methods of designing intelligent computer systems;
- an intelligent user interface;
- training subsystems for designing intelligent computer systems, including a subsystem for conducting a dialogue with the developer and the user;
- a subsystem for testing and verification of intelligent computer systems, including a subsystem for testing the compatibility of the developed system with other systems;
- an information security support subsystem for the intelligent computer system.

The key component of the technology for intelligent systems design is a component design that is represented as a *library of reusable components* and the corresponding *tools for supporting component design of intelligent computer systems*. With its help, it is possible to effectively implement the typical subsystems to support the design of intelligent computer systems.

II. Analysis of existing approaches to solving the problem

The problem is the lack of accessibility and integration in artificial intelligence technologies, which have high initial qualification requirements, lack a unified semantic compatibility solution, and have a high degree of dependency on platforms, subject areas, and components, leading to long development times, high maintenance costs, and difficulties in reusing and integrating previously developed components in new systems.

Existing approaches to solving the problem include component libraries and package managers of programming languages and operating systems, as well as separate systems and platforms with built-in components and means for saving created components.

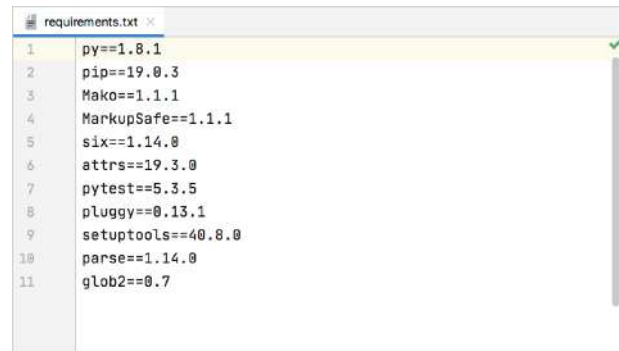
The components of the library may be implemented in different programming languages (which leads to the fact that for each programming language different libraries are developed with their own solutions to various common situations), and may be located in different places, which leads to the fact that the library needs a tool to find components and install them.

Modern package managers such as *npm*, *pip*, *paqt*, *maven*, *poetry* and others have the advantage that they are able to resolve conflicts when installing dependent

components, but they do not take into account the semantics of components, but only install components by the [8] identifier. Libraries of such components are only a repository of components, without taking into account the purpose of components, their advantages and disadvantages, areas of application, the hierarchy of components and other information necessary for the intellectualization of component design of computer systems. Searching for components in *component libraries* corresponding to these package managers is reduced to searching by component identifier. Modern package managers are only "installers" without automatic integration of components into the system. Also a significant disadvantage of the modern approach is the platform dependency of components. Modern component libraries are oriented only to a certain programming language, operating system or platform.

The *pip* package manager is a package management system that is used to install packages from the Python Package Index, which is some library of such packages. Pip is often installed with Python. The pip package manager is used only for the Python programming language. It has many functions for working with packages:

- installation of a package;
- installation of a package of a specialized version;
- deletion of a package;
- reinstallation of a package;
- display of installed packages;
- search for packages;
- verification of package dependencies;
- creation of a configuration file with a list of installed packages and their versions;
- installation of a set of packages from a configuration file.



```
requirements.txt
1  py==1.8.1
2  pip==19.0.3
3  Mako==1.1.1
4  MarkupSafe==1.1.1
5  six==1.14.0
6  attrs==19.3.0
7  pytest==5.3.5
8  pluggy==0.13.1
9  setuptools==40.8.0
10 parse==1.14.0
11 glob2==0.7
```

Figure 1. pip configuration file

The pip package manager works well with dependencies, displays unsuccessfully installed packages, and also displays information about the required package version in case of conflict with another package. An example of a pip package configuration file is shown in Figure 1.

Another example of a package manager is *npm*. npm is a package manager for the JavaScript language. The

npm package manager has a component library (npm Registry) and a command-line user interface. The source code for the package manager and related npm tools can be found at <https://github.com/npm>. The most commonly used npm commands are:

- initializing the project (creating the package.json file);
- install all packages from the package.json file;
- install a package by name;
- deleting a package by name;
- check for obsolete packages;
- display help;
- view installed packages;
- search for packages;
- update packages.

The component approach to the design of computer systems can be implemented within various languages, platforms, and applications. Let us consider some of them.

The ontology implemented in *OWL* (Web Ontology Language) is a set of declarative statements about the entities of the dictionary of a subject domain (discussed in more detail in [9]). *OWL* assumes the concept of an “open world”, according to which the applicability of subject domain descriptions placed in a specific physical document is not limited only to the scope of this document — the contents of the ontology can be used and supplemented by other documents adding new facts about the same entities or describing another subject domain in terms of this one. The “openness of the world” is achieved by adding a URI to each element of the ontology, which makes it possible to understand the ontology described in *OWL* as part of a universal unified knowledge.

The *IACPaaS platform* is designed to support the development, management, and remote use of applied and instrumental multi-agent cloud services (primarily intelligent ones) and their components for various subject domains [10].

The *IACPaaS platform* supports:

- the basic technology for the development of applied and specialized instrumental (intelligent) services using the basic instrumental services of the platform that support this technology;
- a variety of specialized technologies for the development of applied and specialized instrumental (intelligent) services, using specialized platform tool services that support these technologies.

The *IACPaaS platform* also does not contain means for a unified representation of the components of intelligent computer systems and means for their specification and automatic integration.

Based on the analysis carried out, it can be said that at the current state of development of information technologies, there is no comprehensive library of reusable se-

mantically compatible components of computer systems and corresponding component management tools. Thus, it is proposed to implement a library and an appropriate component management tool that will implement seamless integration of components, ensure semantic compatibility of systems and their components, and significantly simplify the design of new systems and their components.

III. Proposed approach

Within this article, it is proposed to take the *OSTIS Technology* [11] as a basis, the principles of which make it possible to implement a library of semantically compatible components of intelligent computer systems and, accordingly, provide the ability to quickly create knowledge-driven systems using ready-made compatible components.

The systems developed on the basis of the *OSTIS Technology* are called *ostis-systems*. The *OSTIS Technology* is based on a universal method of semantic representation (encoding) of information in the memory of intelligent computer systems, called an *SC-code*. Texts of the *SC-code* (sc-texts) are unified semantic networks with a basic set-theoretic interpretation, which allows solving the problem of compatibility of various knowledge types. 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 memory that stores *SC-code* constructions is called semantic memory, or *sc-memory*.

Within this article, fragments of structured texts in the *SCn code* [12] 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.

The basis of the knowledge base within the *OSTIS Technology* is a hierarchical system of subject domains and ontologies.

In order to solve the problems that have arisen in the design of intelligent systems and libraries of their reusable components, it is necessary to adhere to the general principles of the technology for intelligent computer systems design, as well as meet the following requirements:

- ensuring compatibility (integrability) of components of intelligent computer systems based on the unifying representation of these components;
- clear separation of the process of developing formal descriptions of intelligent computer systems and the

process of their implementation according to this description;

- clear separation of the development of a formal description for the designed intelligent system from the development of various options for the interpretation of such formal descriptions of the systems;
- availability of an ontology for component design of intelligent computer systems, including (1) a description of component design methods, (2) a model of a *library of reusable components*, (3) a model of a *specification of reusable components*, (4) a complete *classification of reusable components*, (5) a description of means for interaction of the developed intelligent computer system with *libraries of reusable components*;
- availability of *libraries of reusable components of intelligent computer systems*, including component specifications;
- availability of means for interaction of the developed intelligent computer system with libraries of reusable components for installation of any types of components and their management in the created system. The installation of a component means not only its transportation to the system (copying sc-elements and/or downloading component files) but also the subsequent execution of auxiliary actions so that the component can operate in the system being created.

Based on this, in order to solve the problems set within this article, it is proposed to develop the following system of subject domains and corresponding ontologies:

- Subject domain of reusable ostis-systems components
- Subject domain of a library of reusable ostis-systems components
- Subject domain of the manager of reusable ostis-systems components

IV. Concept of reusable component of ostis-systems

The *Subject domain of reusable ostis-systems components* describes the concept of a reusable component, the classification of components, and their general specification. This subject domain allows creating new and specifying existing components to add them to the library.

As a *reusable ostis-systems component*, a component of some ostis-system that can be used within another ostis-system is understood (see [13]). This is a component of the ostis-system that can be used in other ostis-systems (*child ostis-systems*) and contains all those and only those sc-elements that are necessary for the functioning of the component in the child ostis-system. In other words, it is a component of some *maternal ostis-system*, which can be used in some child ostis-system. To include a reusable component in some system, it must be installed in this

system, that is, all the sc-elements of the component should be copied into it and, if necessary, auxiliary files, such as the source or compiled component files. *Reusable components* must have a unified specification and hierarchy to support compatibility with other components. The compatibility of *reusable components* leads the system to a new quality, to an additional extension of the set of problems to be solved when integrating components.

reusable ostis-systems component

```

:= [typical ostis-systems component]
:= [reused ostis-systems component]
:= [reusable OSTIS component]
:= [ostis-systems ip-component]
:= frequently used sc-identifier*:
   [reusable component]
⊂ ostis-system component
⊂ sc-structure

```

The requirements for *reusable ostis-systems components* inherit the common requirements for the design of software components and also include the following ones [14]:

- there is a technical possibility to embed a reusable component into a child ostis-system;
- a reusable component should perform its functions in the most general way, so that the range of possible systems in which it can be embedded is the widest;
- compatibility of a reusable component: the component should strive to increase the level of negotiability of ostis-systems in which it is embedded and be able to be automatically integrated into other systems;
- self-sufficiency of components, that is, their ability to operate separately from other components without losing the appropriateness of their use.

In the *Subject domain of the library of reusable ostis-systems components*, the most common concepts and principles are described, which are valid for any library of reusable components. This subject domain allows building many libraries, each of which will be semantically compatible with any other built according to the proposed principles. Such libraries store components and their specifications for use in child ostis-systems. An example of a specification of a reusable ostis-systems component is shown in Figure 2.

Versions for the full contents of the *Subject domain of reusable ostis-systems components* and the *Subject domain of the library of reusable ostis-systems components* are represented in the work [15].

The *manager of reusable ostis-systems components* is the main means of supporting component design of intelligent computer systems built by the *OSTIS Technology* ([16]). It allows installing reusable components in ostis-systems and controlling them. The *Subject domain*

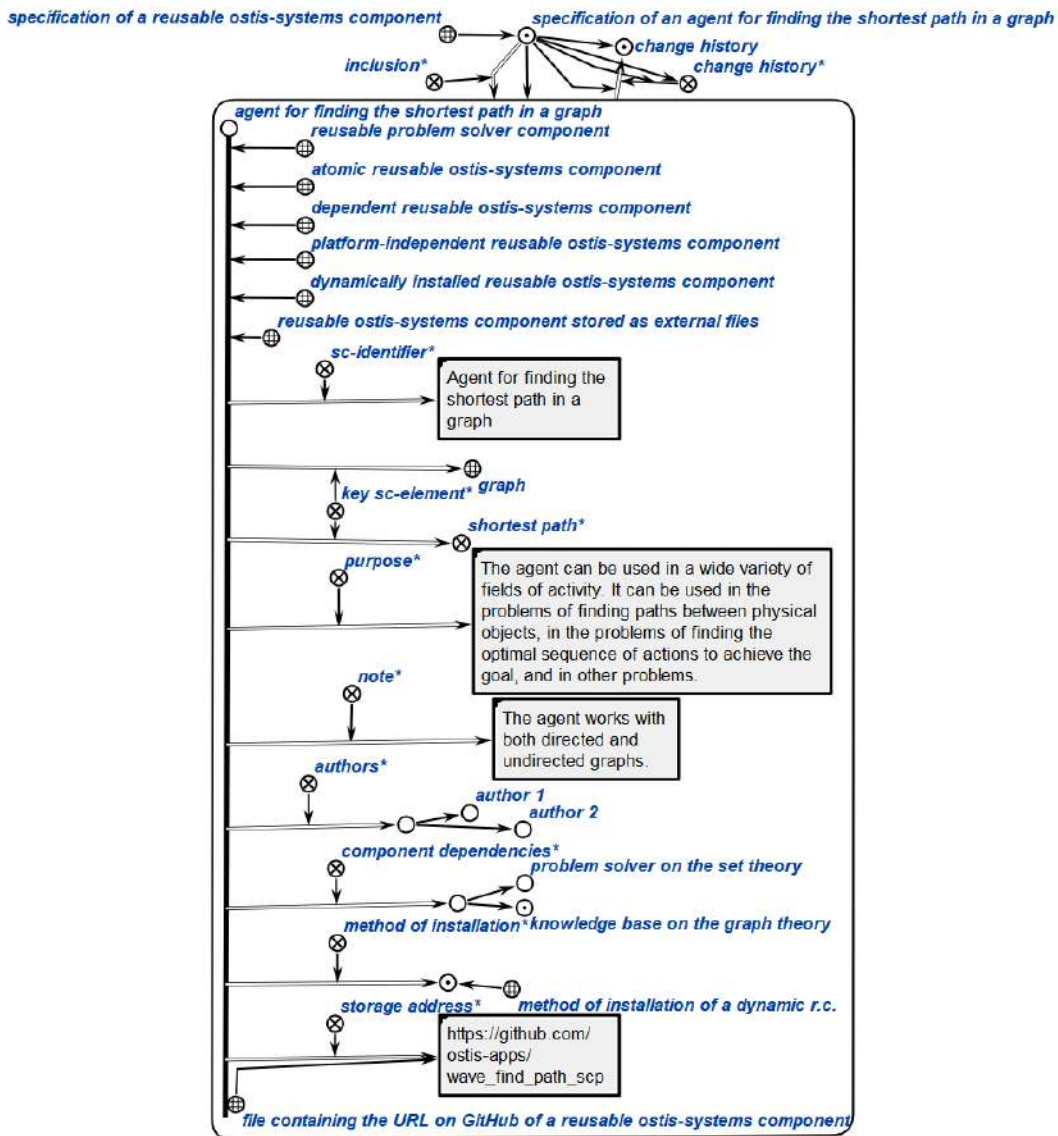


Figure 2. An example of a specification of a reusable ostis-systems component

of the manager of reusable ostis-systems components contains the full specification for the manager of ostis-systems components, the requirements for the component manager, its functionality, the specification of the implementation option for the manager of ostis-systems components, including the sc-model of the knowledge base, the problem solver, and the interface.

V. Architecture of component manager and library of reusable components of ostis-systems

To install reusable components of ostis-systems it is necessary to have a special subsystem in the system: a manager of reusable components of ostis-systems. The component manager interacts with the user and with the library of reusable components. To clarify the specifics

of such interaction, diagrams are developed and SC-constructions necessary to initiate actions when working with the component manager are depicted.

Fig. 3 shows the entity-relationship diagram for the component manager, describing the current state of the component manager functionality.

The diagrams (Fig. 3 and Fig. 4) use the following concepts:

- of entities
 - Developer — a developer (of their local system);
 - OSTIS Metasystem Developer;
 - Component manager — manager of reusable components of ostis-systems;
 - Library of components;
 - System — user's system;

- Reusable component — Reusable component of ostis-system;
- (Reusable) component specification — Specification of reusable component of ostis-system;
- Storage (GitHub) — A repository of components and specifications, such as GitHub;
- Other library — a third-party library of reusable components.
- relationship
 - update;
 - use;
 - subsystem;
 - search;
 - connect;
 - store;
 - link;
 - installation.
- attributes
 - OSTIS Metasystem library — OSTIS Metasystem library of reusable components of ostis-system;
 - OSTIS Metasystem manager — OSTIS Metasystem manager of reusable components of ostis-system;
 - search arguments — search arguments for reusable components
 - * Author — the author of the component;
 - * Class — the class of the component;
 - * Identifier — the name of the component;
 - * Explanation — explanation of the component.

Component Library is a library of reusable components of ostis-systems, which is a subsystem of them. The library's *knowledge base* is a repository of reusable component specifications, and the library also provides an interface to visualise and manage component specifications of the user's system.

Component Manager — is a reusable component manager for ostis systems that is a subsystem for installing, downloading, and tracking components and their specifications for both the user's system and other systems that store reusable components.

The entity-relationship diagram for the component manager from the point of view of the ostis-system user on the example of the OSTIS Metasystem Library (Fig. 3) contains the following information.

The developer uses some ostis-system, a subsystem of which is a reusable component manager and optionally a library of reusable components. The developer can update a reusable component from the OSTIS Metasystem Library using the OSTIS Metasystem Reusable Component Manager. The developer can use the **component manager** to search for components in the OSTIS Metasystem and third-party libraries known to the manager by criteria such as component author, class, identifier, and component explanation fragment. The developer can

connect to other component libraries. The developer can also install the found components into his system.

The entity-relationship diagram for a component library depicts the main relationships between a system, in this case the OSTIS Metasystem, and its subsystems (*manager* and *library*) in terms of the storage of components and their specifications.(Fig. 4). The diagram contains the following information.

OSTIS Metasystem has a subsystem in the form of a library and a component manager. The OSTIS Metasystem library stores many specifications of reusable components. Since all components are stored via GitHub, the manager uses the links provided in the component specifications to access them. The component specifications have a link to the repository that stores the component itself.

Updating reusable OSTIS Metasystem components in the OSTIS Metasystem Library is done through the OSTIS Metasystem Manager and the GitHub repository. The manager allows you to select the required versions of components and install the corresponding component specifications in the OSTIS Metasystem Library. According to these specifications, users using the OSTIS Metasystem will be able to learn about components and install them in their systems.

A component repository such as GitHub has many repositories, each of which can store any number of components and their specifications for installation on other systems using the reusable component manager.

VI. Reusable components installation process

Let's consider the functions of the manager of reusable components of ostis-systems.

reusable ostis-system components manager

```

:= [component manager]
⇒ functions*:
{
• reusable component installation
  ⇒ partitioning*:
    {
    • scnitem
      scnitem component download
    • setting component dependencies
    • translating component scs files into system sc-memory
    }
• search for specifications of reusable components
• downloading specifications of reusable components
}
  
```

In general, component installation consists of the following steps:

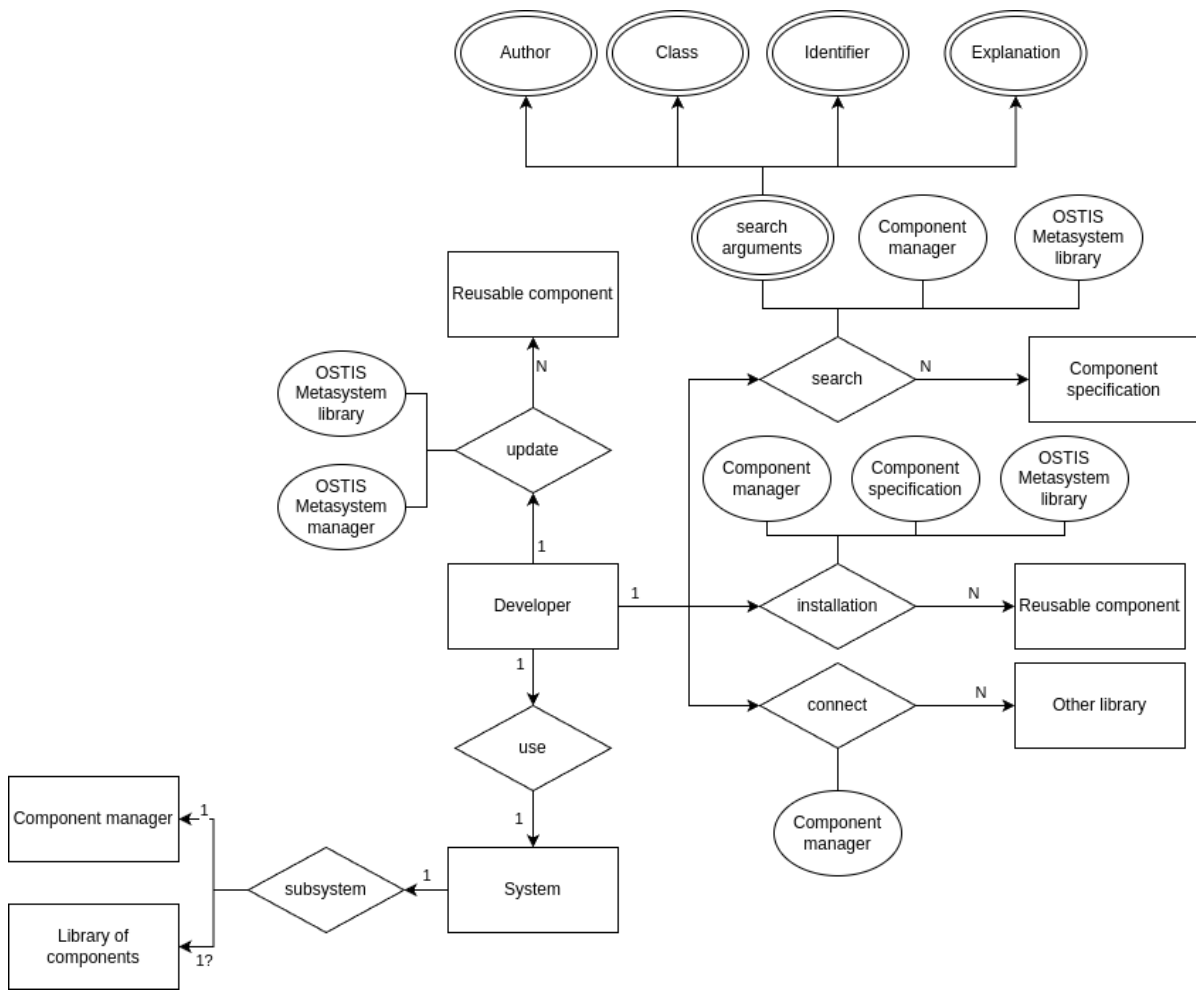


Figure 3. Entity-relationship diagram for a component library

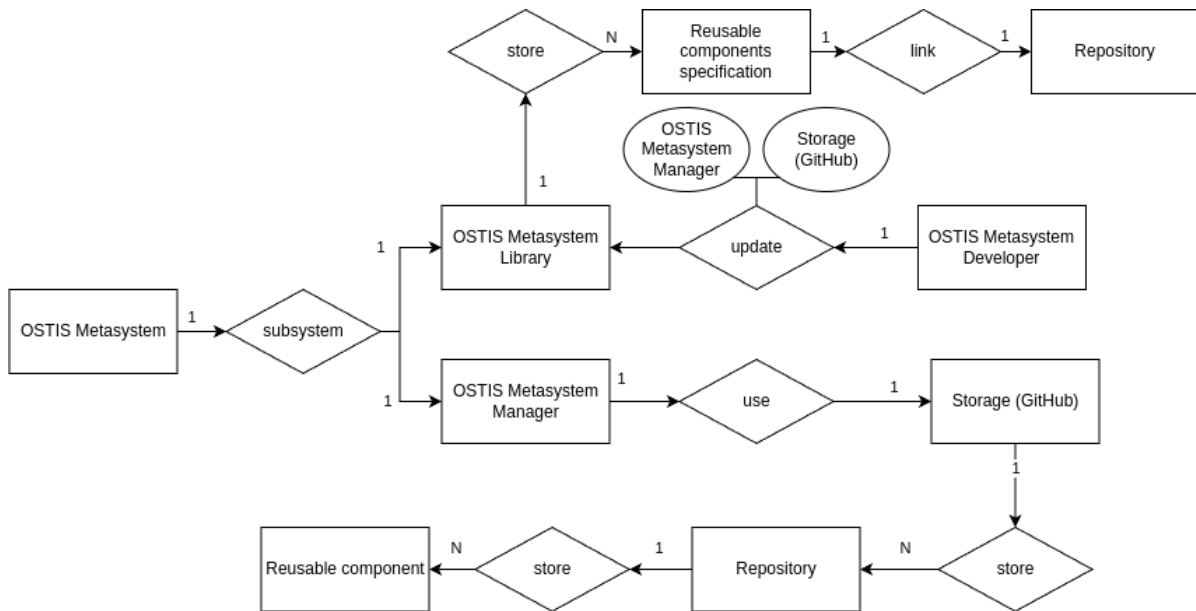


Figure 4. Entity-relationship diagram for a component library

- initiation of the agent to install all component specifications described in the knowledge base;
- to initiate the agent to search for the required component specifications in the knowledge base;
- initiating the agent to install the selected components.

After the user has initiated the component specification agent, the component manager will search the component specifications for references to the component specification repository. The specification file is called *specification.scs* and is stored in the folder with the reusable component itself. If the component manager was able to locate this file, it will load the file into sc memory. The component specification may include:

- identifier of the component;
- classes to which the component belongs;
- indicating the author of the component;
- indicating a note for the component;
- specifying how the component is installed;
- specifying the location (link) where the component is stored.

After the specifications are set, the user can search for components or install them.

The design of initiating the action of searching for component specifications is shown in Fig. 5.

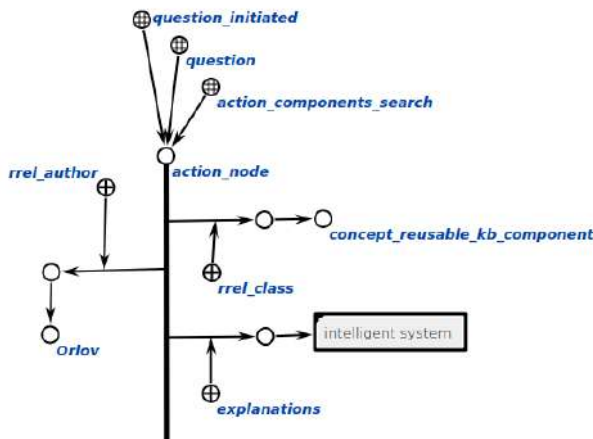


Figure 5. Example of calling the reusable components specification search agent

Three parameters are possible for the agent to search for component specifications: class, author, note. According to the above example, the manager will search for specifications of components created by *Orlov M.K.*, belonging to the class *multiple-used knowledge base component* and having the substring "intelligent system" in the note.

For the agent to find all components known to the system, then *class of reusable ostis-system components* must be passed as a parameter, and then the agent will

find all specifications of reusable components stored in the system.

In order to install a component, you need to pass it as a parameter when calling the component installation agent. The agent will find the required component and its semantic neighbourhood that specifies the storage location of the component and how to install it, then the agent will install the reusable component in the ostis-system.

The design of the component installation agent initiation is shown in Fig 6:

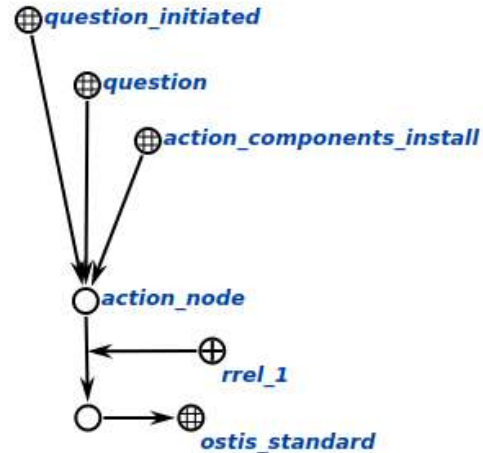


Figure 6. Example of calling the reusable components installation agent

Thus, the component manager and reusable component library allow systems to create and design intelligent systems based on off-the-shelf solutions, thus enhancing system interoperability and simplifying system development.

VII. Specification of ostis-system generation

Component-based design of computer systems means not only extending the functionality of a system already created in some form, but also creating an entire system "from scratch".

For the generation of ostis-systems the manager of reusable components of ostis-systems is used, which provides the possibility to assemble the system from the components available from the libraries of reusable components of ostis-systems.

The following typical sequence of user actions is used to generate ostis-systems.

generation of ostis-systems

- := [creation of ostis systems]
 ⇒ *generalised sequence of user actions**:
- *search for ostis platform*
 - *installing the ostis platform*
 - *search for generic subsystems*

- *installing generic subsystems*
 - *search for reusable components of ostis-systems*
 - *installing reusable components*
 - *configure ostis system*
- := [configuration of the ostis system]
-)

Whether installing reusable components into an already created system or creating a system from scratch, constructs are created in the ostis-system knowledge base to denote which components are installed into the system. Figure 7 shows an example of a construct specifying which components are installed in an ostis system.

Finding and installing an ostis-platform is necessary because different ostis-platforms may be suitable for different classes of tasks and components to be installed in the generated system.

By installing generic subsystems, the functionality of the ostis-system being generated can be greatly expanded. The OSTIS Metasystem Library contains many subsystems often used in other ostis-systems. Typical subsystems include, for example, the subsystem for collective design of ostis-systems, natural language interface, training subsystem, security subsystem and others.

Thanks to the extensive search functionality of reusable ostis-system components, it is possible to search for any components according to various criteria and combinations thereof.

Customising an ostis-system implies setting parameters to specify the peculiarities of the system operation, as well as specifying which users are administrators, developers, experts, and users of the created ostis-system.

In addition to user actions when creating an ostis-system, the ostis-system generation subsystem also registers the created ostis-system in the OSTIS Metasystem. Thus, the OSTIS Metasystem is able to monitor and update the status of the components of this system.

VIII. User interface of component manager and library of reusable components of ostis-systems

The multi-component manager for ostis-systems has a console interface. The component manager is connected to sc-memory as a dynamic component, so it does not require a restart, and you can immediately see the installed components in a running system.

Let's look at the commands with which you can use the component manager. Each command calls the corresponding agent. Agent-based architecture allows you to implement any user interfaces for the component manager of ostis-systems. Any variant of the component manager user interface creates sc-constructs in sc-memory that are needed to invoke the corresponding sc-agent.

action. Set the specifications of reusable ostis-system components

⇒ *agent**:
[ScComponentManagerInitAgent]
⇒ *command to initiate an action**:
[components init]

action. Find specifications for reusable ostis components

⇒ *agent**:
[ScComponentManagerSearchAgent]
⇒ *command to initiate action**:
[components search]
⇒ *possible flags**:
• [author]
• [class]
• [explanation]

When using the search command with the author flag, you must list the system identifiers of the sc nodes that denote the authors of the reusable component. The class flag is used to pass the class name to the component manager to search for components belonging to this class. The explanation flag is used to specify a natural-language fragment that is a substring of the component's explanation. If multiple search flags are listed, components that satisfy all search criteria simultaneously will be found. If you use the component search command without flags, all components whose specifications are downloaded will be found.

action. Install reusable ostis components

⇒ *agent**:
[ScComponentManagerInstallAgent]
⇒ *command to initiate action**:
[components install]
⇒ *flags**:
• [idtf]

The component install command requires the mandatory idtf flag, which the component manager uses to search by system ID for the components to be installed and create the necessary construct to call the component install agent.

The interface of the reusable component library is graphical (Fig. 8). It displays the components that are in the library and provides access to search, browse, and install them.

IX. Example of usage of library of reusable components of ostis-systems

As an example of a library of reusable components of ostis-systems for consideration of an example of work, let's take the OSTIS Metasystem Library.

Installation of the OSTIS Metasystem is performed using the following command sequence.

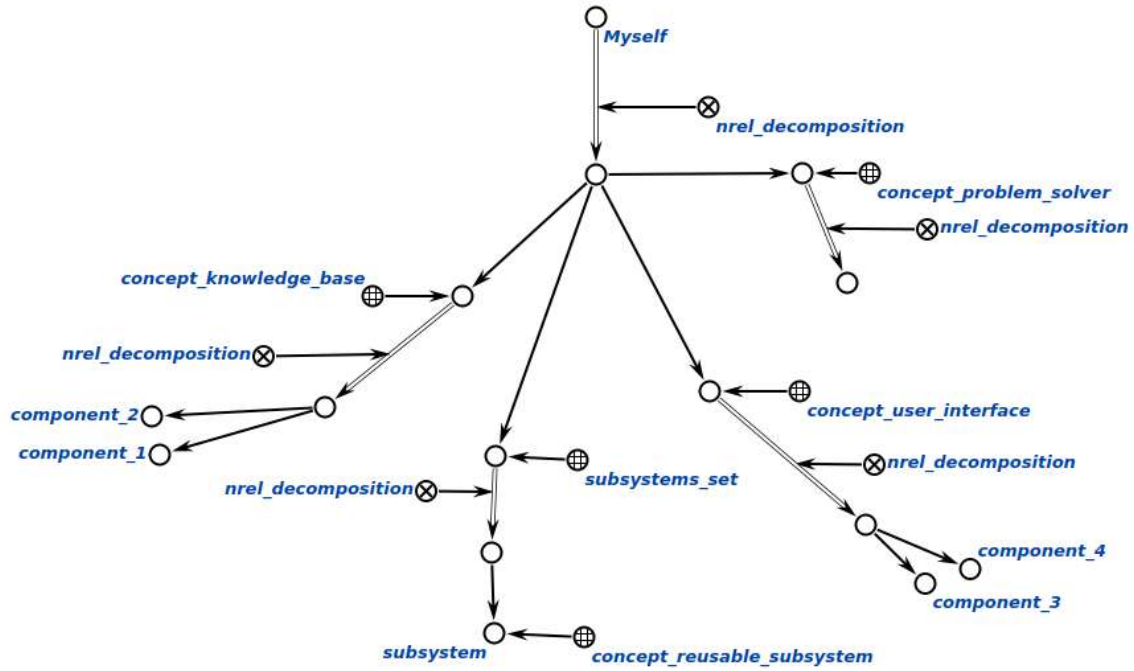


Figure 7. Formalisation of installed components to the ostis-system

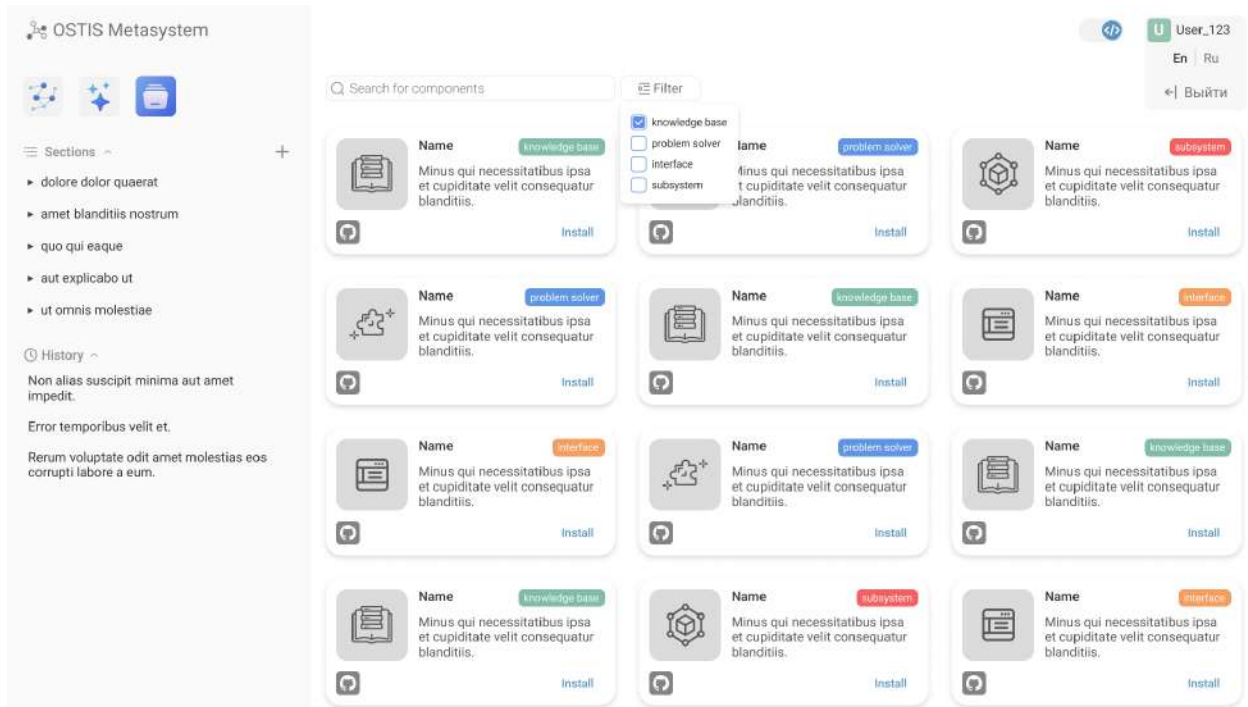


Figure 8. User interface of a library of reusable components of ostis-system

OSTIS Metasystem

⇒ *installation stages**:

- (• *Repository cloning*
 - ⇒ *terminal command**:
[git clone https://github.com/ostis-ai/ostis-metasystem]
- *Change directory to the project root*
 - ⇒ *terminal command**:
[cd ostis-metasystem]
- *OSTIS Metasystem installation*
 - ⇒ *terminal command**:
[./scripts/install_metasystem.sh]
- *Run sc-component-manager*
 - ⇒ *terminal command**:
[./scripts/run_sc_component_manager.sh]

⇒ *component installation procedure**:

- (• *Install all the reusable components specifications*
 - ⇒ *command**:
[components init]
- *Reusable components searching*
 - ⇒ *command**:
[components search]
- *Installation of reusable component*
 - ⇒ *command**:
[components install --idtf <identifier>]

⇒ *usage examples**:

- *Creation of the kernel*
- *Extension of kernel functionality*

Creation of the core

⇒ *stages**:

- (• *Install all the reusable components specifications*
 - ⇒ *sc-agent**:
[ScComponentManagerInitAgent]
 - ⇒ *command to call an agent**:
[components init]
 - ⇒ *result**:
[All the reusable components specifications from OSTIS Library are installed.]
- *Search reusable user interface components*
 - ⇒ *sc-agent**:
[ScComponentManagerSearchAgent]
 - ⇒ *command to call an agent**:
[components search --class concept_reusable_interface_component]
 - ⇒ *result**:
[All the reusable user interface

components are found.]

- *Install sc-models of user interface interpreter*

⇒ *sc-agent**:
[ScComponentManagerInstallAgent]
⇒ *command to call an agent**:
[components install --idtf sc_web]
⇒ *result**:
[sc-web is installed by specification.]
⇒ *note**:
[If you start the web interface after this step, only the start page will load, because the knowledge base is currently empty.]

- *Searching for Knowledge Base components*

⇒ *sc-agent**:
[ScComponentManagerSearchAgent]
⇒ *command to call an agent**:
[components search --class concept_reusable_kb_component]
⇒ *result**:
[Received all components for which their specification states that they are Knowledge Base components.]

- *OSTIS Standard installation*

⇒ *agent**:
[ScComponentManagerInstallAgent]
⇒ *command to call an agent**:
[components install --idtf ostis_standard]
⇒ *result**:
[OSTIS Standard is installed]
⇒ *note**:
[If you now start the web interface again, this step in the web interface will display a page with part of the standard to navigate to.]

)
⇒ *result**:
[The kernel is installed.]

Extension of kernel functionality

⇒ *stages**:

- (• *Search for all available Knowledge Base components in the library*
 - ⇒ *sc-agent**:
[ScComponentManagerSearchAgent]

- ⇒ *command to call an agent**:
[*components search --class concept_reusable_kb_component*]
 - ⇒ *result**:
[Found all Knowledge Base components whose specifications have been installed]
 - *Installing the Knowledge Base component*
 - ⇒ *sc-agent**:
[ScComponentManagerInstallAgent]
 - ⇒ *command to call an agent**:
[*components install --idtf part_polygons*]
 - ⇒ *result**:
[A Knowledge Base component in the form of subject domain of polygons was established.]
 - ⇒ *note**:
[After performing this step, we can find the concept "multiple" in the web interface and browse its semantic neighbourhood. But it is worth noting that the subject domain of triangles, which is a private subject domain of polygons, is empty.]
 - *Installing the Knowledge Base component*
 - ⇒ *sc-agent**:
[ScComponentManagerInstallAgent]
 - ⇒ *command to call an agent**:
[*components install --idtf part_triangles*]
 - ⇒ *result**:
[The Knowledge Base component is installed in the form of subject domain of triangle]
 - ⇒ *note**:
[After performing this step, we can find the concept "triangle" in the web interface and browse its semantic neighbourhood. It is worth noting that the subject domain of triangles, which is a private subject domain of polygons, is fully described and compatible with the subject domain of polygons.]
 - *Creating two sets of triangles*
 - ⇒ *note**:
[At this step it is necessary to find the class "triangle" in the web-interface, create two sets of triangles and add elements to them.
- It is necessary to specify that the sets and their elements belong to the class "triangle".]
 - ⇒ *example**:
[*triangles_1 = {ABC, CDE, XYZ}, triangles_2 = {MNK, CDE, XYZ}*]
 - ⇒ *note**:
[After performing this step, you can check that no operations on sets can be performed now. This can be verified by right-clicking on the node "triangles_1".]
 - *Search for all available problem solver components in the library*
 - ⇒ *sc-agent**:
[ScComponentManagerSearchAgent]
 - ⇒ *command to call an agent**:
[*components search --class concept_reusable_ps_component*]
 - ⇒ *result**:
[Found all components of the problem solver whose specifications are installed.]
 - *Installing the components of the problem solver*
 - ⇒ *sc-agent**:
[ScComponentManagerInstallAgent]
 - ⇒ *command to call an agent**:
[*components install --idtf agent_of_finding_intersection_of_sets*]
 - ⇒ *result**:
[A problem solver component for finding the intersection of two sets is established.]
 - ⇒ *note**:
[After this step, you can check that you can now perform an operation on sets. In the web interface, search for the concept "installed components" and select the node of the desired agent *agent_of_finding_intersection_of_sets*) and run the set intersection agent using the example of two previously created triangle sets. The intersection of the two sets will be found. But it should be noted that this way of launching the agent is long and inconvenient.]
 - *Search for all available interface components in the library*

- ⇒ *sc-agent**:
[ScComponentManagerSearchAgent]
 - ⇒ *command to call an agent**:
[components search --class concept_reusable_interface_component]
 - ⇒ *result**:
[Found all interface components whose specifications have been downloaded.]
 - *Installing the user interface component*
 - ⇒ *sc-agent**:
[ScComponentManagerInstallAgent]
 - ⇒ *command to call an agent**:
[components install --idtf menu_of_agent_of_finding_intersection_of_sets]
 - ⇒ *result**:
[Installed an interface component for an agent to find the intersection of two sets.]
 - ⇒ *note**:
[After this step, the intersection finder can be invoked using a button in the interface, which is much faster and more convenient than the first method. This can be checked by calling the agent to find the intersection of two sets using the example of triangle sets (triangles_1 and triangles_2).]
 - *Setting a logical formula component*
 - ⇒ *sc-agent**:
[ScComponentManagerInstallAgent]
 - ⇒ *command to call an agent**:
[components install --idtf lr_about_isosceles_triangle]
 - ⇒ *result**:
[Established a component with a logical formula for determining whether a triangle is isosceles or not.]
 - ⇒ *note**:
[If you go to the web interface after performing this step, create the necessary fragment for the geometry logic formula parcels and try to run the logic output, it fails because the logic output component is missing.]
 - *Setting the logic inference component*
 - ⇒ *sc-agent**:
[ScComponentManagerInstallAgent]
- ⇒ *command to call an agent**:
[components install --idtf scl_machine]
 - ⇒ *result**:
[Logic inference machine is installed.]
 - ⇒ *note**:
[After performing this step go to the web-interface, create the necessary fragment to send a logical formula on geometry and try to run the logical output, then the formula will generate the necessary fragment of the Knowledge Base. However, this is still not very convenient.]
- *Installing the user interface component*
 - ⇒ *sc-agent**:
[ScComponentManagerInstallAgent]
 - ⇒ *result**:
[Interface component for logic output component installed]
 - ⇒ *note**:
[After performing this step in the web interface after creating the necessary fragment to send the formula on geometry, you can easily call the logical output agent through the interface component.]
-)
- ⇒ *result**:
[The functionality of the system is extended. A system capable of logical inference and finding intersection of sets is obtained. The system has interface components corresponding to these agents. The Knowledge Base on geometrical figures (polygons and triangles) is also obtained.]

X. Conclusion

The component approach is key in the technology of designing intelligent computer systems. At the same time, the technology of component design is closely related to the other components of the technology of designing intelligent computer systems and ensures their compatibility, producing a powerful synergetic effect when using the entire complex of private technologies for designing intelligent systems. The most important principle in the implementation of the component approach is the semantic compatibility of reusable components, which minimizes the participation of programmers in the creation of new computer systems and the improvement of existing ones.

To implement the component approach, in the article, a library of reusable compatible components of intelligent computer systems based on the *OSTIS Technology* is proposed, classification and specification of reusable ostis-systems components is introduced, a component manager model is proposed that allows ostis-systems to interact with libraries of reusable components and manage components in the system, the architecture of the ecosystem of intelligent computer systems is considered from the point of view of using a library of reusable components.

At the moment the manager of reusable components of ostis-systems with console user interface and the library of reusable components of ostis-systems with graphical user interface have been implemented. The subject areas necessary for the implementation of component design have been implemented, and diagrams showing the details of the use and operation of the component manager and the component library have been implemented.

The results obtained will improve the design efficiency of intelligent systems and automation tools for the development of such systems, as well as provide an opportunity not only for the developer but also for the intelligent system to automatically supplement the system with new knowledge and skills.

References

- [1] K. Yaghoobirafi and A. Farahani, "An approach for semantic interoperability in autonomic distributed intelligent systems," *Journal of Software: Evolution and Process*, vol. 34, 02 2022.
- [2] Natalia N. Skeeter, Natalia V. Ketko, Aleksey B. Simonov, Aleksey G. Gagarin, Irina Tislenkova, "Artificial intelligence: Problems and prospects of development," *Artificial Intelligence: Anthropogenic Nature vs. Social Origin*, 2020.
- [3] Olena Yara, Anatoliy Brazhehev, Liudmyla Golovko, Liudmyla Golovko, Viktoriia Bashkatova, "Legal regulation of the use of artificial intelligence: Problems and development prospects," *European Journal of Sustainable Development*, 2021.
- [4] J. Waters, B. J. Powers, and M. G. Ceruti, "Global interoperability using semantics, standards, science and technology (gis3t)," *Computer Standards & Interfaces*, vol. 31, no. 6, pp. 1158–1166, 2009.
- [5] M. Wooldridge, *An introduction to multiagent systems*, 2nd ed. Chichester : J. Wiley, 2009.
- [6] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph processing on GPUs," *ACM Computing Surveys*, vol. 50, no. 6, pp. 1–35, Nov. 2018. [Online]. Available: <https://doi.org/10.1145/3128571>
- [7] P. Lopes de Souza, W. Lopes de Souza, and R. R. Ciferri, "Semantic interoperability in the internet of things: A systematic literature review," in *ITNG 2022 19th International Conference on Information Technology-New Generations*, S. Latifi, Ed. Cham: Springer International Publishing, 2022, pp. 333–340.
- [8] Blähser, Jannik and Göller, Tim and Böhmer, Matthias, "Thine — approach for a fault tolerant distributed packet manager based on hypercore protocol," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2021, pp. 1778–1782.
- [9] V. Torres da Silva, J. S. dos Santos, R. Thiago, E. Soares, and L. Guerreiro Azevedo, "OWL ontology evolution: understanding and unifying the complex changes," *The Knowledge Engineering Review*, vol. 37, p. e10, 2022.
- [10] V. Gribova, L. Fedorischev, P. Moskalenko, V. Timchenko, "Interaction of cloud services with external software and its implementation on the IACPaaS platform," pp. 1–11, 2021.
- [11] Vladimir Golenkov and Natalia Guliakina and Daniil Shunkevich, *Open technology of ontological design, production and operation of semantically compatible hybrid intelligent computer systems*, V. Golenkov, Ed. Minsk: Bestprint [Bestprint], 2021.
- [12] (2022, September) IMS.ostis Metasystem. [Online]. Available: <https://ims.ostis.net>
- [13] Shunkevich D.V., Davydenko I.T., Koronchik D.N., Zukov I.I., Parkalov A.V., "Support tools knowledge-based systems component design," *Open semantic technologies for intelligent systems*, pp. 79–88, 2015. [Online]. Available: <http://proc.ostis.net/proc/Proceedings%20OSTIS-2015.pdf>
- [14] G. Sellitto, E. Iannone, Z. Codabux, V. Lenarduzzi, A. De Lucia, F. Palomba, and F. Ferrucci, "Toward understanding the impact of refactoring on program comprehension," in *29th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2022, pp. 1–12.
- [15] M. K. Orlov, "Comprehensive library of reusable semantically compatible components of next-generation intelligent computer systems," in *Open semantic technologies for intelligent systems*, ser. Iss. 6. Minsk : BSUIR, 2022, pp. 261–272.
- [16] —, "Control tools for reusable components of intelligent computer systems of a new generation," in *Open semantic technologies for intelligent systems*, ser. Iss. 7. Minsk : BSUIR, 2023, pp. 261–272.

ТЕКУЩЕЕ СОСТОЯНИЕ СРЕДСТВ АВТОМАТИЗАЦИИ КОМПОНЕНТНОГО ПРОЕКТИРОВАНИЯ OSTIS-СИСТЕМ

Орлов М. К., Макаренко А. И.,
Петрочук К. Д.

В работе рассматривается подход к проектированию интеллектуальных систем, ориентированный на использование совместимых многократно используемых компонентов, что существенно сокращает трудоемкость разработки таких систем. Ключевым средством поддержки компонентного проектирования интеллектуальных компьютерных систем является предложенный в работе менеджер многократно используемых компонентов.

Received 03.03.2024