*Bull. Nov. Comp. Center*, Comp. Science, 48 (2024), 13–32


# On some technological issues of LuNA active knowledge system implementation*

K. Buyko, R. Kapralova, M. Kopylov, A. Kudryavtsev,
V. Perepelkin, A. Pirozhkov

**Abstract.** Active knowledge concept is a methodology for parallel programs construction automation. It allows automatic construction of programs which meet non-functional requirements in particular subject domains. To support the automation an active knowledge base has to be constructed, which contains information on peculiarities of the subject domain. LuNA system is an academic project aimed at implementation and practical research of the active knowledge concept. In the paper we concern some technological problems which arise during practical implementation of the active knowledge concept, as well as their solutions developed in LuNA project.

**Keywords:** active knowlege concept, automated program construction, parallel program.

## Introduction

Development of high performance numerical parallel programs is a complex problem, which requires skills and knowledge in both applied domain and system parallel programming. Automation of programs construction is advantageous, because automated program construction requires less qualification and effort. Also, the quality of automatically constructed programs is potentially higher than that of manually constructed programs (hereinafter by quality of a program we mean its efficiency in terms of execution time, memory consumption, network load, etc.). The same can be seen from the history of conventional compilers, where the quality of constructed machine code was poor, but now it is far beyond practical abilities of an assembly language programmer.

Automatic programs construction is an algorithmically hard problem. This is confirmed by the fact that no universal automatic programs constructor exists today. Only particular solutions exist, which are capable of constructing programs for a particular class of problems.

Among many works devoted to programs construction automation in the paper we consider the active knowledge concept [1], which is an approach

*The study was carried out under state contract with ICMMG SB RAS FWNM-2022-0005.

to automatic programs construction. With this approach, to provide high quality of automatically constructed parallel program in particular subject domain, an active knowledge base has to be constructed. Active knowledge base is a machine-oriented representation of knowledge about a subject domain, which allows automatic construction of high quality parallel programs. A program is constructed automatically from its high level specification for a particular class of problems in the subject domain.

The active knowledge concept is based on the theory of computational models based synthesis of parallel programs [2]. Implementation of the active knowledge concept in practice raises a number of technological issues to resolve. In the paper, we consider an active knowledge system prototype to concern the issues. The prototype is based on LuNA parallel programs construction system [3], and further development of the prototype will produce the subsequent version of the system.

The paper is organized as follows. In Section 2, the main ideas are explained and a whole prototype overview is given. In the next sections particular technological issues are studied by discussing particular components of the system. In the end of the paper a conclusion is given.

## 1. Active knowledge concept ideas and terms

The key idea, which provides automatic programs construction is the idea of solving a problem that is formally defined based on a computational model. In details the idea is explained in [2] (see also [4]). Here we will briefly explain it with some non-essential simplifications.

Computational model is a bipartite oriented finite graph whose parts are two sets of vertices, called the set of operations and the set variables correspondingly. The arcs entering and exiting an operation define which variables are the input and output variables of the operation, respectively. A computational model describes a certain subject domain, where the properties of objects in the domain are represented by the set of variables (property values are represented by variable values), and the ability to compute the values of some properties from the values of some other properties is represented by the set of operations (see the example in Figure 1).

For example, in the "trigonometry" subject domain the following computational model can be defined. Variables represent the lengths of the sides of a triangle, the magnitude of its angles, the radius of inscribed circle, etc. Operations represent the ability to compute, for example, the area of the triangle from two sides and the angle between them; the ability to compute the angle from the other two angles, etc.

To provide the ability to compute values, a computational module called a code fragment must be provided. Conventional subroutine is a good example of code fragment.

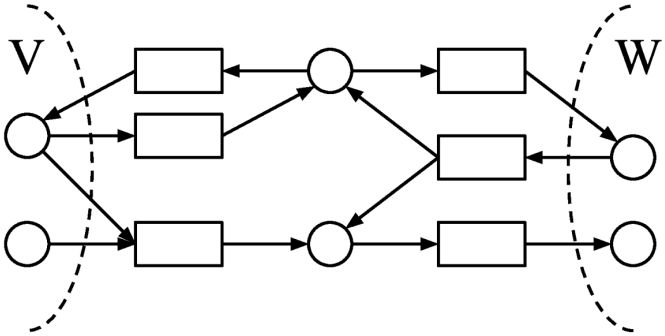


**Figure 1.** An example of a computational model.
Circles and rectangles denote variables and operations correspondingly

In order to construct a program based on computational model a problem to be solved by the program has to be defined. For that two subset of variables V and W are defined. The set V defines input variables of the program while W defines output ones. It is called VW-task. If there is a subset of operations in the computational model, the orderly execution of which allows obtaining the values of variables in W provided the values of variables in V are given, then the subset of operations is called VW-plan. Any VW-task may have zero or more VW-plans. VW-plan represents a VW-task solution algorithm.

Once a VW-plan is obtained, there is not a problem to invoke code fragments, corresponding to the operations, step by step computing values of variables until all variables in W are computed.

This is the basic idea that makes it possible to automatically construct programs which solve new problems using existing code fragments.

In the trigonometry example, once we have included all the trigonometry formulas as operations into the computational model, we are able to automatically solve a wide variety of trigonometry problems by defining different V and W sets, including the problems which can only be solved in multiple steps.

Note, that in a computational model there are normally more operations than in a VW-plan. Also, while in one VW-task a variable can be input, in another VW-task the same variable can be output, intermediate or unused.

Implementation of the idea of automatically constructing a program based on a computational model, in practice requires consideration of a number of issues, such as optimization of program's non-functional properties, provision of program's dynamic properties, data formats conversion, code fragments representation formats, execution organization, data storing, profiling, etc. Active knowledge base can be defined as an aggregate of several components, the main of which is a computational model, and

other components are code fragments, non-functional properties specifications, execution profiles and other entities, needed to construct a program fully automatically.

Concerning the technological issues of the active knowledge system implementation the following system architecture can be considered. The active knowledge system is a software, capable of automatic programs construction based on an active knowledge base. Input for the system is a specification of a VW-task for a given computational model, values of variables in V and non-functional requirements for the program to construct. The output of the system is either a program which solves the VW-task, or values of variables in W, depending on which mode is selected – program generation or problem solution.

The system is an aggregate of a number of components, which can be combined in various configurations. Some components can operate on their own or in connection with other components, while other components can only operate in connection with other components. The basic components connection scheme is the microservice approach, where each component is a web service with a HTTP REST-like interface. However, a more tight API-based connection is also possible. The components are as follows.

**Interpreter.** A component, which receives a computational model, a VW-task and values of input variables, and returns the values of output variables. The component implements the problem solution mode of the system. The interpreter derives a VW-plan and performs operations execution. The interpreter can work as a standalone component, or use external executor, values storage, planner. It can be accessed via graphical user web-interface or a command line interface.

**User Interface.** A web-based graphical user interface which allows a user to describe computational models, browse variables and values, define VW-tasks, inspect executing processes, etc. The user interface can interact with many system components, giving access to their functionality to the user.

**Generator.** A component, which receives a computational model and a VW-task and produces a program, which solves the task. So, there are two main possibilities for a user to solve a VW-task. The first one is to use interpreter, and the second one is to generate a solver and run it.

**Code library.** A storage which provides access to code fragments for other components, such as interpreter or executor. It encapsulates technical peculiarities of code fragments and provides a common interface to use them.

**Planner.** A component which derives VW-plans from VW-task on given computational model. Deriving a plan is an optimization problem, because different VW-plans possess different non-functional properties, and depending on the optimization criteria different VW-plans should be derived. The planner is mainly used by interpreter and generator.

**Profiler.** A component which gathers and analyzes the execution characteristics of constructed programs. The gathered information can be used by interpreter or planner to improve efficiency.

**Values storage.** A component which stores values of variables in the long term or during a particular execution process. It is mainly used by user interface, interpreter and executor. Generator normally constructs a program which uses ordinary memory (shared or distributed memory of a multi-threaded or multi-process program), but it is also possible to generate a program which uses an external storage, such as values storage.

**Executor.** A component capable of executing a code fragment by applying it to given input arguments in order to produce output arguments. Since generated programs can also be considered as code fragments, executor can run generated programs too.

**Core.** A component, which connects all the components together encapsulating network details and particular implementation details of the components. It is also responsible for organizing execution of complex commands, which involve multiple steps and components (e.g. solving a user-defined VW-task by generating and running a solver program on an external computing cluster).

Other components may exist, but their consideration is out of the scope of the paper.

## 2. User interface

To work with the active knowledge system, a user interface is required. The interface of the active knowledge system is a user web interface for interactive interaction with active knowledge bases and programs generated and executed on them. Figure 2 shows the general appearance of the interface; an elemental computational model describing a linear dependence was specially created for demonstration.

The interface should provide users with access to all the functional capabilities of the system and hide the complex internal organization of the system from the user. For the interface of the active knowledge system, clarity is important to facilitate the user's understanding of the concepts of the system and the correct distribution of his attention when working with a relatively new concept.

The interface of the active knowledge system has a number of features that make its development a non-trivial task, the solution of which requires the use of scientific methods. Let us consider these features.

The interface should take into account the features of human work with the active knowledge system. It is necessary to study how the work with the active knowledge base is organized, what standard scenarios and actions are included in the process. Users may require different ways of working with

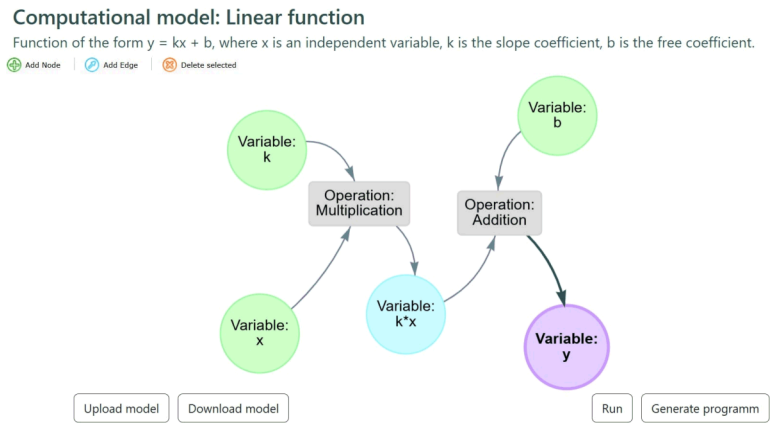


**Figure 2.** User interface of the Active Knowledge System

the system in different subject areas, including ways of navigating the active knowledge base and ways of displaying elements of computational models and data.

The interface must be flexible and adapt to changes, since the project is at an early stage of development.

Graphical notation is necessary to display elements of the active knowledge base and the program execution process. Graphical notation must support user customization. As the system develops, the graphical notation must be easily expanded with new operators, such as variable collections and mass operators.

The active knowledge base can be large, then it will be difficult to navigate through it; in addition, a person is not able to perceive a large number of concepts at a time. Therefore, it is important to introduce interface computational models that contain only the elements necessary to set a task for the system.

To generate a program, the user needs to define its interface. It is also not enough to select input and output variables, one needs to assign a format to them, that is, determine how the program will work with arguments. The system also provides the ability to generate programs in accordance with non-functional requirements. The list of interfaces, argument formats and requirements that the system can work with will change as the system develops.

The interface supports interaction with the computational process to display its stage and interact with it.

**General scenario of user interaction with the system.** Consider the process of user interaction with the system. The user navigates the active knowledge base, presented in the form of computational models. The user

edits the computational models; loads variable values and selects code fragments; sets tasks for the system, determining which elements are input and which are output.

Now the user can start the computational process for the task, track its progress, pause or cancel its execution. The user can also request the generation of the corresponding program, download it and run it on user's device.

**Current version of the user interface of the active knowledge system.** The interface displays computational models. Graphic notation specially represents operations and variables; variables with and without values; operations with a given and unspecified code fragment; input, output and unmarked variables.

The computational model can be imported or exported in JSON-based format or created using the computational model editor, which provides the ability to add and delete variables, operations, and relationships between them.

Variables and operations are handled using dialog boxes. They display the properties of the selected computational model's element and elements for working with values and code fragments for variables and operations respectively.

The variable value can be set by the user by uploading a file to the system, or it can be deleted. The value available in the system can be downloaded.

The code fragment for the operation is selected by the user from the list of code fragments available in the system. It is possible to search for a computational model by name and description. There are elements for determining the correspondence between the arguments of the computational model and the variables of the computational model.

To inform the user about events occurring in the system, the interface has a stack of pop-up messages. At the moment, the functionality for starting the program generation and executing the computational process for the task has not been implemented.

## 3. Code generator

The process of program construction can be viewed as the mapping of a given algorithm (in the form of a VW-plan) onto available computational resources and the specification of the control structure necessary for the constructed program that solves the problem specified by the user.

The process of program construction is viewed as a sequential decision-making process. At each step of this process, different decisions can be made. For example, choices may include selecting a node for performing an

operation, determining the order of operations, or deciding how to allocate data in memory. The decisions made can affect non-functional properties of the constructed program. For instance, they can impact the execution speed of the generated program, memory consumption, and the utilization of other computer resources.

Automatic program construction is a challenging problem. It is algorithmically hard to choose a good solution from a variety of possible options. This obstacle impedes the practical applicability of automatic program construction, as algorithms do not always succeed in finding a good solution. The possibility of human intervention in the construction process not only helps to mitigate these limitations but also significantly expands the range of tasks that can be addressed through automatic construction. Humans can leverage their knowledge to make decisions in situations where construction algorithms may struggle.

Customizable construction involves the user being able to influence the decision-making process. This allows the user to impact the program construction at a high level without relying on low-level development tools. In this problem statement, a person can make partial decisions while leaving the remaining decisions to the construction system. This approach ensures an effective distribution of decision-making between the automatic construction system and the user.

For example, the user can specify the desired non-functional properties (such as minimizing program execution time or making it energy-efficient) or select the necessary operations in the computational model that they consider essential for effectively solving the problem. The generator in this process understands the conditions set by the user and selects suitable operations from the provided computational model (knowledge base). In this way, a program is constructed that aligns with the user's requirements.

Thus, customizable program construction combines two approaches: automatic construction and construction with user influence. This approach is capable of yielding better results in tasks where system algorithms fail to provide satisfactory quality.

To implement customizable program construction, it is first necessary to implement automatic program construction. In this process, it is essential to ensure that the user can manage the program construction. Let us consider the key elements and approaches through which the generator implements the program construction process.

Program construction is carried out using special "building materials" — constructs. Each construct contains the necessary parameters and properties for the program construction.

Several types of constructs are used for the generator: program specification, memory cell, port, snippet and partially defined program.

Program specification is the input representation through which the user describes the program to construct. The user specifies which variables are input for the program and which are output, as well as the operations that the user wants to be performed and some other information.

A memory cell is a data storage unit used in the process of computation. In the generated program, memory cells are the variables that are used in operations.

To correctly link variables between operations, ports are used. Each port in an operation describes a variable, as a variable can be either an input or an output for the operation. This allows the correct order of operations to be established according to the information dependencies in the constructed program.

Operations in the program are described by the main building blocks – snippets. Snippets are responsible for executing one or several operations in the program. Moreover, snippets not only describe the operations that need to be performed in the program, but they can also carry out additional tasks, such as providing communications between processes if the user requires the construction of a parallel program. Additionally, these snippets can be used for input and output of data from external sources (files, command-line arguments), or for data transformation to correctly pass them to the ports, or other operations.

All the aforementioned constructs are used within the main construct — partially defined program. This construct serves as the skeleton of the future generated program.

The process of program construction occurs iteratively. The entire program is constructed within the "partially defined program" construct. Initially, this construct contains only one snippet that terminates the program, as the construction begins from the end of the program. However, it is also possible to start constructing from the very beginning or from several snippets, regardless of whether it is the start or the end of the program. It all depends on the implementation of the construction algorithm.

For example, if the user wants to construct a program that calculates the area of a triangle given two sides and the angle between them (Figure 3), the construction process will begin with the partially defined program containing the last snippet for outputting a variable to a file. This means that the snippet will be responsible for writing the computed area of the triangle to a text file.

In each iteration, the construction algorithm checks the ports of the snippets in the partially defined program and refers to the program specification to identify which operations can fit the ports in the partially defined program. It adds suitable snippets and the necessary memory cells. Thus, in each iteration, the process of refining the partially defined program occurs. The constructed partially defined program becomes fully defined when the

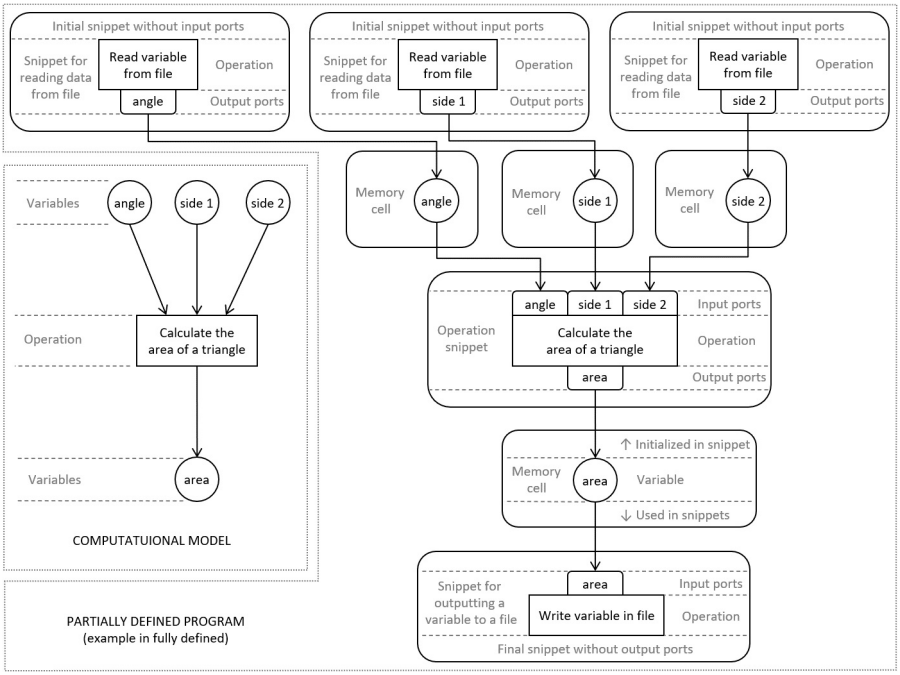


**Figure 3.** Example of calculating the area of a triangle using a computational model and a partially defined program

required snippets from the program specification have been utilized, and a correct order from the input variables to the output variables has been fully established in accordance with the information dependencies.

In the partially defined program that calculates the area of a triangle, the construction algorithm examines the snippet for outputting the variable to a file to see which input ports are available. In this case, there is only one port for the variable representing the area of the triangle. Next, it is necessary to review the program specification to identify which operations have the computed area of the triangle as their output port. For the sake of simplicity in this example, let's assume that there is only one operation in the program specification, which has three input ports: the angle and the two adjacent sides of the triangle corresponding to that angle, and one output port for the area of the triangle. This snippet is then added to the partially defined program. Additionally, a memory cell is added to store the variable for the computed area of the triangle. This represents a step in the refinement of the program. Then, snippets for reading data from file will be similarly added, which will then be fed into the input ports of the snippet that performs the area calculation. At this point, all the necessary constructs for calculating the area of the triangle are in place.

As soon as the partially defined program is fully defined, code generation takes place. Each construct contains the necessary data that allows it to be transformed into code in C++. Thus, the program is generated.

The user can customize the construction process by selecting which operations are more preferable in the input representation. The construction algorithm will immediately add snippets with these operations to the partially defined program, allowing the user to make decisions on behalf of the construction algorithm. As a result, the generator will not have to choose.

For example, the user knows many parameters of the triangle: all three sides and some angle. Therefore, to calculate the area, two operations may be suitable: one based on the three sides or the other based on two sides and the angle between them. The user can specify the desired operation in the program specification at their discretion, and the generator will automatically complete the remaining snippets. Alternatively, the user can choose not to make any decisions and leave the decision-making process to the generator.

Let us consider how a task is formulated in the active knowledge system and how it appears in the described generator. The task is formulated using a VW-task and a computational model. The computational model describes the variables and operations involved. In the generator, it is necessary to list these operations in the input representation and to describe the ports–specifying which variables are input for each operation and which are output. From the VW-task, it is needed to define which variables are inputs to the program and which are outputs from the program. The user also specifies certain non-functional properties in the input representation, thereby implements customizable program construction.

Thus, the generator can perform customizable program construction using active knowledge technologies.

## 4. Profiler

Program profiling is an integral part of software development. Its essence lies in collecting the program's characteristics during its execution. These characteristics may include execution time, errors encountered during runtime, or communication duration (in the case of distributed computations). The collected information can be used to analyze program performance and optimize it. For instance, the generated profile can help identify the causes of performance degradation and take measures according to the SLOW methodology [5] or eliminate imbalances across computational nodes [6]. The specifics of the active knowledge system provide extensive opportunities for both profiling and applying profiling information. For instance, the use of a computational model enables profiling of operations and variables, while the planner's generation of VW plans allows it to utilize profiling in-

formation for more efficient plan construction. A key feature of profiling in the active knowledge system is the ability to conduct it with consideration of the domain area. This approach enables embedding knowledge about profiling specifics tailored to the unique characteristics of the domain into the system.

A profile obtained from a single computation is valuable on its own. It can help identify errors, debug the computational model, detect imbalances in processor core loads, and redistribute operations more efficiently. However, multiple profiles collected from computations of the same computational model provide even greater practical value. They enable the calculation of statistics, such as the average execution time of an operation or the average size of an output variable. Additionally, they allow the determination of the dependency between the execution time of an operation or the size of its output variable and the input data. All this information can be presented directly to the user to make necessary changes for optimization and debugging or shared with other components of the active knowledge system. For example, the planner can use it to create more efficient VW-plans, the executor for load balancing, and the variable storage for storage optimization.

In addition to providing raw profiling data or statistics, it is also possible to deliver estimates. Most estimates are functions of two arguments: pre-collected information about the estimate target and information from the current computation. For example, the planner may request not just the average execution time of an operation but an estimation of this time for generating a new VW plan. The profiler, in turn, provides this estimate based on the average execution time of the operation, the nature of its dependency on input variable sizes (pre-collected data), and the size of the input variables (current computation data). The quality of this estimate depends on the completeness of the provided data and the algorithms implemented by the profiler.

Providing profiling information and its derivatives to other components of the active knowledge system allows them to improve their operation. It is essential to note certain limitations associated with this. For instance, profiling information obtained for one computational model is generally not applicable to optimizing another, even if they share some similarities (e.g., certain operations map to identical code fragments). The same applies to different configurations of the computing system. Essentially, the computational model is optimized for a specific system, and transitioning to another configuration may require restarting the profiling process.

Profiling information can be collected in several ways. The simplest and most straightforward approach is to collect the profile every time the user runs a computation. However, if the task is known for the computational model and all necessary data for its resolution is available (e.g.,

input variables have values, and operations are assigned code fragments), the computation can be run without user involvement for profiling purposes. Moreover, additional tasks not previously defined by the user can be generated, executed, and profiled.

Currently, the "Profiler" component has been implemented to parse log files, calculate statistics from the collected profiling data, and store them in a PostgreSQL database. The component can be deployed in a Docker container for easier integration into various environments. The "Executor" component is instrumented with code that collects profiling data into log files. Additionally, a REST API has been developed for retrieving profiling information (reports) and providing estimates (Figures 4 and 5 correspondingly).

For these API, an attachment mechanism ("localStorage" block) has been implemented, allowing additional information to be attached to requests for use by the profiler in calculations. This optimizes the component's

```
{
    "reports": [
        {
            "common": {
                "computationId": "comp://localStorage/1"
            },
            "data": [
                {
                    "eventType": "OPERATION_START",
                    "id": "sum",
                    "timestamp": "2024-10-22T00:00:00.000Z"
                },
                {
                    "eventType": "OPERATION_STOP",
                    "id": "sum",
                    "timestamp": "2024-10-22T00:00:01.000Z"
                }
            ]
        }
    ],
    "localStorage": {
        "comp": {
            "1": {
                "computationalModelId": "cm://localStorage/1"
            }
        },
        "cm": {
            "1": {
                "otherLocations": [
                    "cm://remoteStorage3/111"
                ],
                "operations": {
                    "sum": {
                        "moduleId": "mod://localStorage/1",
                        "inputs": [
                            "x",
                            "y"
                        ],
                        "outputs": [
                            "z"
                        ]
                    }
                }
            }
        },
        "mod": {
            "1": {
                "otherLocations": [
                    "mod://remoteStorage1/123",
                    "mod://remoteStorage2/321"
                ],
                "processorType": "CPU"
            }
        }
    }
}
```

**Figure 4.** Report API example

```
{
    "estimates": [
        {
            "common": {
                "type": "OPERATION_EXECUTION_TIME_PREDICTION",
                "computationId": "comp://localStorage/1"
            },
            "data": [
                {
                    "operationId": "sum1"
                },
                {
                    "operationId": "sum2"
                }
            ]
        }
    ],
    "localStorage": {
        "comp": {
            "1": {
                "computationalModelId": "cm://localStorage/1"
            }
        },
        "cm": {
            "1": {
                "otherLocations": [
                    "cm://remoteStorage3/111"
                ],
                "operations": {
                    "sum1": {
                        "moduleId": "mod://localStorage/1"
                    },
                    "sum2": {
                        "moduleId": "mod://localStorage/2"
                    }
                }
            }
        },
        "mod": {
            "1": {
                "otherLocations": [
                    "mod://remoteStorage1/111",
                    "mod://remoteStorage2/111"
                ],
                "processorType": "CPU"
            },
            "2": {
                "otherLocations": [
                    "mod://remoteStorage1/222",
                    "mod://remoteStorage2/222"
                ],
                "processorType": "GPU"
            }
        }
    }
}
```

**Figure 5.** Estimate API example

operation since, in the absence of such information in the request body, the profiler would need to request it from other components, potentially leading to significant communication overhead. This mechanism also enables isolated testing of the profiler from other system components. Furthermore, the API employs an optimization technique where parameters common to each report/evaluation can be moved to a common parameters block.

In future it is planned to eliminate the log file and implement sending profiles directly to the profiler's endpoint (buffering can be used to reduce the number of requests), implement of algorithms for generating evaluations and integration with components that consume evaluations.

## 5. Executor

Program execution is a key stage in the software lifecycle, during which parts of a program's code are executed on a computing platform. This process involves interpreting or compiling the source code, loading the necessary data, and managing the processor, memory, and other system resources. The primary goal of program execution is to ensure correct and efficient operation, which requires considering the capabilities and limitations of hardware resources, such as processor performance, available memory, and network bandwidth.

Modern computational challenges impose increasingly stringent requirements on the program execution process. One such challenge is the execution of parallel programs, which is particularly important for tasks where the volume of data or the computational workload is so extensive that the task can no longer be solved on a single machine. In such systems, individual parts of the program are executed across multiple nodes of a multicomputer, which must interact to achieve a common goal. This necessitates task coordination, data dependency management, and synchronization between nodes, significantly complicating the execution process compared to sequential execution on a single device.

The execution of parallel and distributed programs also entails managing non-functional characteristics, such as scalability, fault tolerance, and energy efficiency. For instance, programs must adapt effectively to changing workloads by redistributing tasks to optimize performance. Additionally, these systems must account for the characteristics of diverse architectures and execution environments to minimize data transmission delays and maximize the utilization of available computational resources. If all decisions regarding the execution order of operations, mapping of operations to multicomputer nodes, and memory management are made during the program generation phase, it becomes impossible to ensure the program's dynamic properties. Therefore, there arises a need for a mechanism to make decisions dynamically during execution.

The LuNA system supports two primary approaches to program execution:

1. Compilation — This approach involves constructing a program based on a pre-planned computation graph and code fragments using the generator component, which produces a single code fragment to be executed. In this case, the program transitions from the original computational model to a final computational model for which an executor is implemented. Often, the final computational model corresponds to the hardware computational model (CPU or GPU). All nondeterminism is resolved during translation, resulting in a program where the entire computational process is strictly defined.

2. Interpretation — This is a method of planning, generating, and executing a program where each code fragment is executed sequentially or in parallel, and the next stage (or plan step) is selected based on the results of the preceding execution. This process is handled by the interpreter component, which implements the original computational model of the program. When the computation process is not strictly determined, the interpreter takes such decisions dynamically during execution. In contrast to the compilation approach, all nondeterminism is resolved at runtime.

Each of these approaches has its own advantages and disadvantages. Dynamic decision-making is more efficient in some cases, while static decision-making works better in others. Consequently, there is a need for a hybrid approach that combines the benefits of both static and dynamic methods.

In the hybrid approach, some decisions are made statically during the translation phase, while others are made dynamically during execution. The efficiency of this hybrid method depends on how the decision-making process is divided. In this sense, the hybrid approach involves forming an intermediate representation of a program. First, the source program is translated into this intermediate model, with some decisions being made statically. Then, this model is executed in interpreter mode, dynamically resolving decisions that were not made statically. To effectively apply the hybrid approach, several issues have to be addressed to determine the most suitable intermediate program representation:

1. What intermediate computational model is most appropriate for a given case?
2. How should the source program be translated into the chosen computational model?
3. Which runtime system should be selected to execute the intermediate representation in the given scenario?

The answers to these questions depend on the specific task to be executed and even on the input data used for the task. Implementing this approach

requires capabilities for resource management, dynamic adaptation of the execution process, and both static (e.g., computational characteristics of the processor or network bandwidth) and dynamic (e.g., processor load or the volume of incoming data) information about the task and its execution process.

Thus, the executor component must meet the following requirements:

- Extensibility — to allow the addition of new types of code fragments for execution.
- Access to information about node characteristics and status — enabling execution that takes this information into account.
- Support for various levels of execution management — to facilitate the operation of other components, such as the interpreter, that make decisions regarding execution.

Additionally, the executor must satisfy certain technical requirements necessary for executing application tasks for which the system is intended. For instance, it must be capable of operating without network connectivity, as such scenarios are encountered in fields like geophysics.

Based on these requirements, a distributed architecture for the executor component was developed. This architecture, illustrated in Figure 6, is built around two key entities: Manager and Machine, each of which performs strictly defined functions.

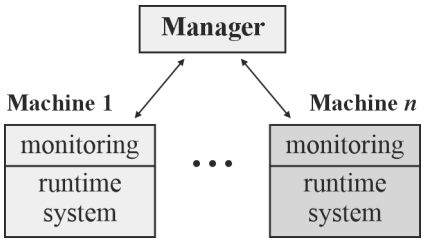


**Figure 6.** Executor architecture

Manager provides an interface for other components of the active knowledge system to interact with the executor. It is also responsible for collecting information about the state of all machines, selecting an appropriate runtime system for task execution, and making decisions related to the execution process. This functional organization allows the Manager not only to abstract the implementation details of different machines but also to make dynamic decisions based on a holistic analysis of the system's state. This capability is particularly important for tasks requiring features such as load balancing.

Machine, in turn, focuses on executing operations at the level of individual multicomputer nodes. It handles node status monitoring, preparation for task execution, and the initiation of operations using the "code fragment library" component. A critical feature of the Machine is the division of its functionality into two parts: monitoring and runtime systems.

- The monitoring part gathers information about the node's state and adapts it for different types of machines, such as personal computers or computing clusters.
- The runtime systems part enables the easy integration and support of new runtime systems, thereby extending the component's capabilities to execute programs on various platforms.

## 6. Core

The core of the system serves as a binding link between all components, creating a unified system and ensuring the operation of the active knowledge base. It enables message exchange within the system, allows for complex decision-making logic regarding the selection of the target request point, and separates specific implementations and interfaces from the abstract representation of requests to components.

The process of the system's operation is divided into several stages, each implemented in a separate component by different people, with each developer bringing their own professional skills and experience. This leads to significant differences in implementation technologies. The integration of components, standardization of protocols, and data transfer formats is a crucial task that requires a solution. Without a binding element and established abstract interfaces for interaction, components will be unable to exchange data with each other or with the user. This work is dedicated to the problem of establishing communication between the system's elements and the user.

To implement the full range of functions for designing and executing parallel programs, several types of components are being developed: user interface, generator, scheduler, variable value storage, and others. They must interact by exchanging requests and information. For executing requests of the type "get/execute/save," additional computations may be required, necessitating navigation across all nodes and effective distribution of computational load. Interactions between components encompass a range of complex aspects, including:

1. Requests to the variable value storage. Since there may be several independent variable storages connected dynamically, it is necessary to address search, aggregation, and distribution tasks among them. Additionally, extra mechanisms may be employed, such as distributed value storage or data duplication to enhance reliability.
2. Executor. When a request for execution is made, a decision must be made about the target executor for that request. Factors such as the capabilities of the executors, their load, reliability, and other parameters are taken into account. Additional processes may also be

utilized, such as launching tasks on multiple executors to obtain results from the first one that completes.

3. Monitoring the number of active tasks on each component, as well as their hardware specifications and availability, is crucial for correctly directing requests. This includes monitoring system resource status and managing task queues, which helps avoid overloads and ensures stable system operation.

The core is a binding element that ensures connection and interaction between components. The main goal here is to develop the system's core for constructing and executing parallel programs within the active knowledge system. The tasks include developing a message exchange protocol, defining message formats, designing the interface, and establishing a request handling approach from clients.

The core's main entity is the request. Requests are sent by components with the content "get/execute/send". Formalizing and simplifying, each request can be represented as a type Get/Set, a name, for example, VAR_VALUE (variable value), and some parameters such as an identifier. A Get request contains only parameters, while a Set request has a body with the data that needs to be sent.

As for the data and request exchange protocols, it has been decided to use HTTP and gRPC with HTTP/2.0. Requests to both the core and other components can be transmitted via either protocol. HTTP is used as a more traditional and familiar communication tool in a multi-service architecture, while gRPC is a new, efficient, and reliable method of message exchange and is experimental. Both servers, HTTP and gRPC, accept requests and package them into internal representations with the information described above.

The request lifecycle in a simplified form looks like: sent by a component; received by the core; the core determines the endpoint of the request; the core opens a data transmission stream between components; the request is executed. The request reception point in the core is handled by the server, after which routing must be performed. This stage involves the main decision-making logic regarding the target node of the request. Decision-making may require additional computations unique to each request. Describing the logic for handling each request within the core is not an optimal solution, as the logic may frequently change, switch, or be developed by other project members. A more appropriate solution would be to separate the decision-making responsibility to external software — the operator. The operator connects to the core as another component, but unlike the other working parts of the system, it does not primarily respond to requests; it only determines the target node.

For the core to pass a request to the operator, the operator must indicate which requests it will handle. To facilitate this, we transition from the concept of a request to an event. Requests are a subset of events. An event is a marker indicating that a request has been received, resolved, completed successfully or with an error, that a request was sent from a component or an operator, etc.

The operator subscribes to the events it is prepared to handle; if the event is the arrival of a request, it responds with the component that should handle the request or directly with the response. Otherwise, the operator may not perform any external actions, using the information for its purposes. For decision-making, the operator may require additional information from other components. The list of components connected to the core is passed along with the event, allowing the operator to address specific components, bypassing the routing stage.

Operators typically exhibit passive behavior, responding to events from the core, but they can also perform active actions. They have several opportunities available: requesting a list of components, redirecting (Get request to component A $\Rightarrow$ Set request to component B), or requesting to a component. Thus, operators open significant possibilities for scaling the system, modifying and expanding behavior through a ready interface and functionality. The logic of the operators can subsequently be formalized into an abstract description for even simpler system behavior modification.

The next stage of the request lifecycle is the creation of the data transmission stream. For this, the core, having received the specific target component of the request, calls its adapter. Adapters facilitate the transition between the component interfaces and the core's internal representation. Through the adapter, request stream handlers are created, connecting with the corresponding ones received from the request's source from the server, enabling data exchange. After data transmission is completed, as in other cases, an event signaling the completion of the request is sent.

Currently, a basic core with adapters for the code fragment library and variable value storage has been implemented. In the current implementation, static routing at the core level is used as the operator. Additionally, a user interface for the system is running on the core side.

The set goal, namely the development of the core of the system for constructing and executing parallel programs, has been achieved. The core functions and can process requests to access the code fragment library and variable value storage. In the future, there are plans to both expand the number of adapters and thus the supported components of the system and to redesign the architecture to support external operators and events.

## Conclusion

In the paper the authors' current understanding of active knowledge system architecture and technological issues is considered. Although it is not final and will evolve further, it was concerned worth sharing, because it covers a wide range of problems which arise when attempting to implement automatic programs construction, and it is concerned how the active knowledge concept allows addressing the problems.

The ideas considered are being implemented as an academic software project LuNA (Language for Numerical Algorithms), so the paper shows a practical view on the topic, not just a theoretical consideration. The software is used to implement practical applications.

## References

[1] Malyshkin V. Active Knowledge, LuNA and Literacy for Oncoming Centuries / Essays Dedicated to Pierpaolo Degano on Programming Languages with Applications to Biology and Security. — 2015. — Vol. 9465. — Berlin, Heidelberg: Springer, P. 292–303.

[2] Valkovsky V.A., Malyshkin V.E. Synthesis of Parallel Programs and Systems on Computational Models / Ed. V.E. Kotov // AN USSR, Sib. branch, Computing Center. — Novosibirsk: Nauka, 1988 (In Russian).

[3] Malyshkin V.E., Perepelkin V.A. LuNA fragmented programming system, main functions and peculiarities of run-time subsystem // Proc. 11th Int. Conf. on Parallel Computing Technologies (PaCT-2011). — 2011. — P. 53–61 (LNCS; 6873).

[4] Malyshkin V.E., Perepelkin V.A. definition of the concept of a program // Problems of Informatics. — 2024. — No. 2. — P. 16–31. DOI: 10.24412/2073-0667-2024-2-16-31. — EDN: CEDVVD (In Russian).

[5] Sterling T., Brodowicz M., Anderson M. High Performance Computing: Modern Systems and Practices. — Morgan Kaufmann, 2017.

[6] Perepelkin V.A. Optimization of fragmented program execution based on profiling // Sixth Siberian Conf. on Parallel and High-Performance Computing: Program and Abstracts. — Tomsk: Tomsk University Press, 2011. — P. 117–122.