Bull. Nov. Comp. Center, Comp. Science, 21 (2004), 99–111

The DCMS library for open architecture applications*

M. Ostapkevich

Abstract. This paper presents a Dynamically Configurable Modular System (DCMS), which is a library for the development of applications with an open architecture. The DCMS is oriented to a wide set of applications, such as mathematical simulation, Web technologies, and so on. By now, the library has been used in the WinALT simulating system, and the knowledge base and the portal development. The substantiation and description of the DCMS architecture are given. A number of applications, based on the DCMS are discussed. Such features as simplicity of maintenance and usage, adaptability for the user's requirements are gaining more importance in order that a complex software be successfully developed. The concept of open software architecture give rise to facilitating the development of software with these features.

1. Introduction

As capabilities of hardware grow, the users require a better quality of application software. Primarily, they expect a more convenient interface, a better interactivity, a longer lifetime and a support of different kinds of hardware with a wide range of performance as well as availability of miscellaneous operating systems. To account these requirements and to successfully develop an application, the following is necessary:

- Easy porting to constantly emerging new platforms and operating systems;
- Easy insertion of new functions into application so as to keep up with the growing user's demands;
- Application porting to distributed computing systems so as to gain a higher performance and reliability.

When developing such applications the following questions arise:

- How the modules structure should be organized to obtain its flexibility, extensibility and ease of maintenance;
- How the communications between remote processes should be organized in a distributed application;
- How to facilitate an application porting to other platforms;
- How to rise an abstraction level up to that required by an application.

*Supported by the Russian Foundation for Basic Research under Grant 03-07-90302.

The solution to these questions could decrease the complexity of the open architecture application construction. The goal of the DCMS creation is to provide a tool for solving these questions.

2. Methods and tools for application program development

The Dynamically Configurable Modular System (further DCMS) appeared as part of the WinALT simulating system development for fine-grain algorithms and structures [1]. In earlier times, such a simulating system was only required to perform simulation by a certain scenario described in a program. Nowadays, the list of requirements is bigger:

- The system must be capable of working with as huge amount of data as required, because it must simulate huge 3D pipelines, optical structures, physical models, and so on.
- The system must have advanced visualization, as no qualitative analysis of a simulating phenomenon can be done without it. The value of simulation as a whole is very doubtful without such an analysis especially because of the structural complexity and data size of practically useful models.
- The data exchange between the system and other applications should exist. This makes it possible to embed the simulation as a phase of data processing in longer chains, where the source data for simulation are borrowed, for example, from an image processing system, and the results are to be sent back to it.
- Standard requirements such as reliability, high performance, flexibility, ease of maintenance or portability are gaining grater importance.

As a result of the influence of these new requirements, the new concepts of software development have appeared.

The concept of distributed and parallel software systems was introduced for increasing the amount of data under processing, improving the reliability and performance. The distributed simulating system [2] can simultaneously be executed at several interconnected hosts. Different hosts can process different parts of model objects under control of a single simulating program (data parallelism). Or, these hosts can perform different phases or aspects of simulation (functional parallelism).

The concept of open software systems [3] was proposed for construction of portable programs with easy maintenance and capability to work cooperatively with other programs. Such software systems must have the following features: extensibility, interoperability, portability and friendliness of the user's interface. Extensibility means ease of a new function addition or function modification without affecting the others. Interoperability is the

ease of communication and cooperation with other programs. Portability denotes the ease of reimplementation of a software system on a new platform. Friendliness of the user's interface is the compliance with the existing user's interface standards, convenience for the user. The user-friendly interface is logical and conceptually homogeneous and all of user's functions are documented in the user's manuals.

Any of the mentioned concepts does not require obligatory tools, libraries or languages. Nevertheless, the complexity of development depends on the adequacy between the tools used and the adopted concepts and methods of development. As a result of WinALT development process analysis and planning of its updating, the idea of the DCMS library has appeared. The goal of the DCMS is to facilitate the design of open software systems.

The DCMS is far from being the first tool for open software systems. Even before the appearance of the very concept of open software systems, many languages (Modula-2 [4]), system tools (linkers) had some limited support of certain aspects of the openness. The most wide spread tools dedicated specially to the openness are COM+ [5] and CORBA [6]. Among others, there is a kind of software named "middleware" (JMQ, MSMQ), coordination languages (Linda, JADE). As samples of a limited support for certain aspects of openness or the full support of openness with narrowed application area these could be mentioned:

1. Extensible Markup Language (XML) metalanguage for data description, whose primary goal is to unify the data representation formats;
2. Remote Procedure Call (RPC) protocol and a set of utilities for the distributed application construction;
3. Common Gateway Interface (CGI) interface for Web/HTTP based on the distributed application construction;
4. ISAPI, Servlet, Java Server Pages (JSP) enhancements of CGI, oriented towards a better performance.

The COM+ and the OLE are, respectively, a model and tools for application assembly if this application is represented by a set of components. The COM+ can be considered as basis of the OLE. The OLE application contains an OLE statically linked library and a set of external modules: standard system components that implement common functions, which are not specific in application, and the user's components with application functions.

The CORBA is a metatool, a technology for the open systems construction. The CORBA consists of the Interface Definition Language (IDL), a request broker, a network protocol for interactions within a distributed application (IIOP) and a set of standard services (naming, lifecycle, event,

object trader, transaction, security, persistency, etc.). A CORBA application is formed by the components, interacting with CORBA and with themselves. Their interface is defined in the IDL. There is a great number of packages that implement the CORBA technology (IANA, ORBit). All of them support the same language and protocols. This makes it possible to communicate between applications based upon different CORBA implementations.

The DCMS just as COM+ and CORBA is a tool for the open software development, but it is not aimed at the same objectives and solutions as in COM+, CORBA. The DCMS is complementary to them. It is intended to solve the problems, which lie at the periphery and have no convenient solutions. What is more, it is planned to implement interoperability with both CORBA and OLE by using their services and components and exporting the DCMS modules as components visible to the COM+ and the CORBA applications.

The principal feature of the DCMS is the orientation towards a mixture of compiled and interpreted modules (half-interpreter system). Such a mixture is quite typical of simulating systems, as it gives a greater flexibility at run time and application adjustability by the user, although it lacks the performance of fully compiled systems. For example, the WinALT interpreter allows defining models with any number of arrays, procedures, imported external libraries without modification of the WinALT itself. Without interpreted language, the WinALT user would lack a convenient way to describe models, and each new model would require a new system assembly instead of its reconfiguration.

The principal means of the interface definition in CORBA based on the IDL is static. The typical CORBA application is a collection of compiled components statically using interfaces of each other and exchanging data with the structure and the types defined at compilation. A typical DCMS application uses a hierarchy of interface definition kinds from fast and static to slower and flexible. Among all of them, the event-driven kind of an interface plays the most important role for application modules. This interface gives a maximum flexibility and reconfigurability of intermodular links, because it allows modifications of the interface at the run time without affecting all of its users that are not interested in such a modification. Also, a DCMS application deals with data of the type and the structure altering at the run time.

3. The DCMS implementation

The DCMS is the library of interface and implementation modules (as is understood in Modula-2 [4]). The division of the module into two parts allows abstracting from its implementation when using it. For example, let

an interface for the associative search data structure be defined. Different implementations based on a list, AVL or RB trees or a hash table can be produced for this interface. If one implementation module is substituted by another, there is no need to modify the source texts that use the respective interface. The goal of the DCMS creation is to facilitate the construction of applications having all the properties of open software systems. The substantiation and description of the architecture of the DCMS first version as well as a model of intermodular cooperation, which is entirely based on events, is given in [7]. The module generates an event in order to execute the represented operation. The modules that have the implementation of the required operation and are subscribed to a generated event, are activated for operation execution. The principal advantage of such a model is the ease of application modifications due to minimization of visible interfaces.

The application with an open architecture must be scalable and extensible. To attain such qualities, one should have a modular structure with the ability to add and to exclude modules at the run time. The usual tasks for such applications are: module management (registration, unregistration, export of their interfaces to other modules) and establishment of intermodular interfaces. The module management in the DCMS is supported by implementation module manager and external module manager. Intermodular communications are based on event-driven interface supported by the event manager and the basic type manager that provides a single format for the printed object representation in memory.

Construction of the distributed processing requires the inter-host data exchange protocols that are based on common data formats. The central role for distributed processing is played by the following DCMS modules: the event manager, the basic type manager, the object manager, I/O method, and the data format support modules.

The degree of portability is mostly determined by the proportion of the platform-dependent and independent modules. The unified system level is introduced so as to decrease the number of platform-dependent interfaces as it gives one interface for all the platforms (supported by the DCMS) instead of a few.

For making data and procedures more concise and readable, the level of their abstraction should be high enough so that objects and relations between them could be described in application terms. This task is accomplished by the basic type manager and the application type manager.

4. The DCMS basic features

The DCMS features that help to attain the goals stated above are discussed below.

4.1. The division of modules into interfaces and implementations. Prof. N. Wirth has proposed such a division in Modula-2 [4]. In Modula-2 program, there is always one implementation for each interface. Later, the idea of interface separation was developed in the COM and the CORBA technologies, which are neutral to programming languages [5], [6]. The COM and the CORBA permit the existence of multiple implementations for the same interface. A similar separation into interface and implementation exists in the DCMS. The novelty in such a separation is introduction of multiple interface definition methods (hierarchy of the interface kinds), which is briefly described in 4.2 and 5.6.

4.2. Hierarchy of the interface kinds. Unlike Modula-2, COM, or CORBA, DCMS provides more than one method of the interface definition. A hierarchy of predefined interface kinds exists. It is also possible to add new interface kinds. The reason for existence of hierarchy is that it would be difficult to use just one kind of interfaces for all software system levels (from system to application). This is because different abstraction levels are required for different system levels. It is convenient to describe interfaces just like the used programming language prescribes at the lowest level. At higher levels, a better flexibility is required and event-driven interfaces are more appropriate. The highest levels deal with the user interface and should be user friendly, understandable and customizable by the user who might want to tune the system to particular needs. The reason for extensibility by the new interface kinds is interoperability with other software systems. They may have their own kinds of interface definition. A good way to obtain interoperability with them is to allow the interface definition just the way they are used in such systems.

4.3. The finegrain modularity. Usually, a module source text is located in one or a few files. There can be some documentation and build-in script files as well. All these kinds of module data (the source text, documentation, scripts for building, installation, uninstallation, archiving and so on) are split to small files. Each file corresponds to a certain aspect. The goal of division into a big number of small files is to improve readability, reusing and maintenance as well as facilitating automatic processing such as compilation, testing, documentation generation or statistics gathering. A slightly more complicated design and coding of a module is fully compensated by the gained quality of the result. The average file size for the DCMS kernel is slightly less than 1Kb. The fine-grain modularity does not have visible performance overheads because it does not increase the number of function calls.

The last peculiarity follows from the three above-mentioned. It is the independence of implementation of the kind of interface used for the interface

definition. A new kind of interface can be supported in the earlier developed implementation. This feature improves implementation reusability even if the kinds of interface are completely changed. But any extension or modification of supported interface kinds may not violate the so called interface invariance. This means that the semantics of operations, which are available through interface, remains the same no matter what interface kind is used.

5. The DCMS architecture

The two main parts of the architecture description are the structure of a system and interoperation between its parts. The structure is described by means of modular decomposition while the interoperation is described as hierarchy of the interface kinds.

5.1. The modular decomposition. The system consists of the kernel with a fixed set of modules and an unlimited number of external modules. The kernel is represented by the following levels: the abstract level, the unified system level, and the service level.

5.2. The abstract level collects implementations for abstract data and control structures. This level is immediately used by the two others. The goal of the level is to decrease the redundancy of other kernel parts. The multiple data used and control structures are extracted into separate modules of this level. Such a separation improves efficiency and reliability of the kernel. Among all others, the central role is played by implementation of the expulsive tree data structure [8], which is used for the associative data search, hierarchical and sparse data representation. The WinALT uses this module in its object manager so as to resolve an object reference by the name and in the substitution manager to keep substitutions found.

5.3. The unified system level. The purpose of this level is to improve the OS platform independence in order to obtain the DCMS kernel portability. Most of the modules that depend on the OS interfaces reside at this level. System property module gives a unified method to obtain and the alter hardware and system settings without dependence on a particular OS. All essential platform parameters are represented as values that can be read and modified. The values are identified by a number. The manager of module implementations registers the kernel modules, manages their state and assists in the search for an implementation for a particular interface. The external library manager provides operations for loading and unloading of the external libraries.

5.4. The service level is central for the kernel. Its modules implement the main features and functions of the kernel. The manager of the basic

types provides the interoperability of DCMS applications by the data format within process space. The manager of formatted I/O implements interoperability for the data stored in the files. The event manager supports an event-driven model of interaction between modules. It provides the means of definition and usage of interfaces for the external modules (XPL). The naming manager serves for the global data naming. This module facilitates the intermodular communications and the user interface construction. The XPL module manager allows loading and unloading of the DCMS external modules (external program libraries) and using the functions implemented within these modules via their interfaces. It is this module that makes it possible to construct applications as a set of modules, thus making the application architecture extensible and scalable.

5.5. External modules. The biggest part of the DCMS functions and all the application functions reside in the external modules. Currently it makes the architecture open. In the future it will be distributed as well. The initialization module serves for application configuration retrieval and loading of the required XPL modules and data files.

The console interface module contains the user text console input/output operations for the atomic type values. The Web/CGI interface module implements communication with the user, based on HTML pages sent upon HTTP and filled-in forms sent back by the user. This module provides visualization of an hierarchical object and contains functions for interoperation with the Apache Web server. The goal of the GUI module is to give a

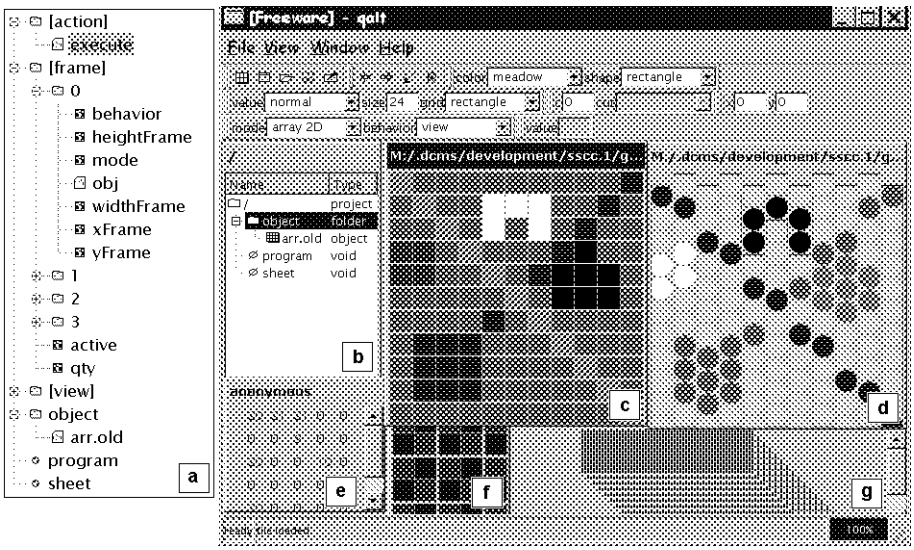

Figure 1

unified means for all supported platforms for the window and the graphical primitive management, the user function invocation, input/output/edit of values. A prototype of the WinALT GUI that is based on the DCMS and the QT (TrollTech), is depicted in Figure 1. Window A contains the full tree of project structure, B shows it partially opened, C and D contain the WinALT object on the rectangular and the hexagon grids, E, F, and G show objects in numeric, Margolus and 3D deck representations.

5.6. Hierarchy of the interface kinds. Different intermodular interactions can be characterized by different important requirements from the maximal performance to the maximal openness. There is no single kind of the interface description that can meet all the requirements. Thus a decision was made to provide multiple kinds of interface description. A set of standard kinds forms a hierarchy. The next level is based upon its ancestor and brings about more openness though slightly degrading a bit in performance.

A function prototype written in a programming language is the lowest level. It is based on the programming language statements for the signatures of interface functions. But many of the languages does not permit separating of the interface and implementation to such a degree that more than one implementation could correspond to the same interface within one application.

The next level in the hierarchy, which is based on the structures of pointers to interface functions, overcomes this fault. Such a kind of interface description is adopted in OLE. The fault of interfaces defined at this level is the difficulty of their modification. After addition of a new function, the whole structure of the interface is changed. Thus, it influences all the procedures that use this interface structure even if they are not intended to use this newly introduced function. One of the ways to resolve this difficulty is to declare a new interface with a new structure each time when a function needs to be added. In fact, this is how it is solved in OLE. There are two inconvenient consequences of this approach: the number of interfaces created to solve the same task grows and the implementation of an interface has the obligation to support not only the latest version of this interface, but all its ancestors used at least in one module. As a result, the program code size grows, and redundant objects are introduced. The alternative to such a solution is given in the next interface kind level.

The level of a dispatcher function is based on the existence of one function (dispatcher) that imitates all the operations located in the implementation. The description of interface at this level consists of operation code declarations and the description of parameter semantics. The same code may denote different operations in different interfaces. Any module that supports this kind of the interface definition must have the dispatcher function with the following prototype:

```
Unsigned f(unsigned *param_array);
```

treat the first element of `param_array` as a code of operation, the second item – as a value for the result; the number and the meaning of the rest of items depend on the operation code and must be treated as is prescribed by their semantics. The benefits of this approach were presented in detail in [9]. Particularly, the problem of insertion of new functions has a convenient solution. An implementation using a certain interface does not "see" new operations if they are not needed. Actually, only the dispatcher function of the respective implementation is modified. The inconvenience of this interface kind level is that one has to know not only the operation code, but also the descriptor of implementation or its entry point to the dispatcher function. A procedure invocation is usually preceded by a descriptor or entry point search. This results in decreased readability and bigger size of the source text.

The level of commands for extensible virtual machine differs from the previous one by the existence of the only dispatcher function (within the application scope) and by globally defined operation codes. It can be said that the dispatcher function executes one command of an extensible virtual machine at a time.

One application has only one interface of the event-driven level. It consists of a set of events that were either generated at least once or had at least one handler at a certain moment. The event-driven level of the interface kinds is most important for the DCMS open architecture application construction. There can be only one reason for deviation from this interface kind, which is performance optimization in critical fragments of a code or in a real-time code.

The user interfaces are those that lie at the top of the hierarchy. This kind of the interface description is oriented to initiation of operations by the application user. There are several examples of different user interfaces: graphical user interface; natural language interface; Web interface; and text console interface.

6. The DCMS application sample

Let us consider a sample that implements the basic Web portal functions: the input of user's requests to the database and the generation of report to a user (Figure 2). The client part is represented by Browser (1) on the user's machine, while the server part is a Web Server (2) and a DCMS-based portal application. The DCMS application consists of the kernel and two external modules: Web Interface Module Ui1W1.xpl (3) and the Request Execution Module Ctrl1Req1.xpl (4). Let us assume that Web server and DCMS application are located at portal.sscc.ru, and the name of application is searchbook.elf.

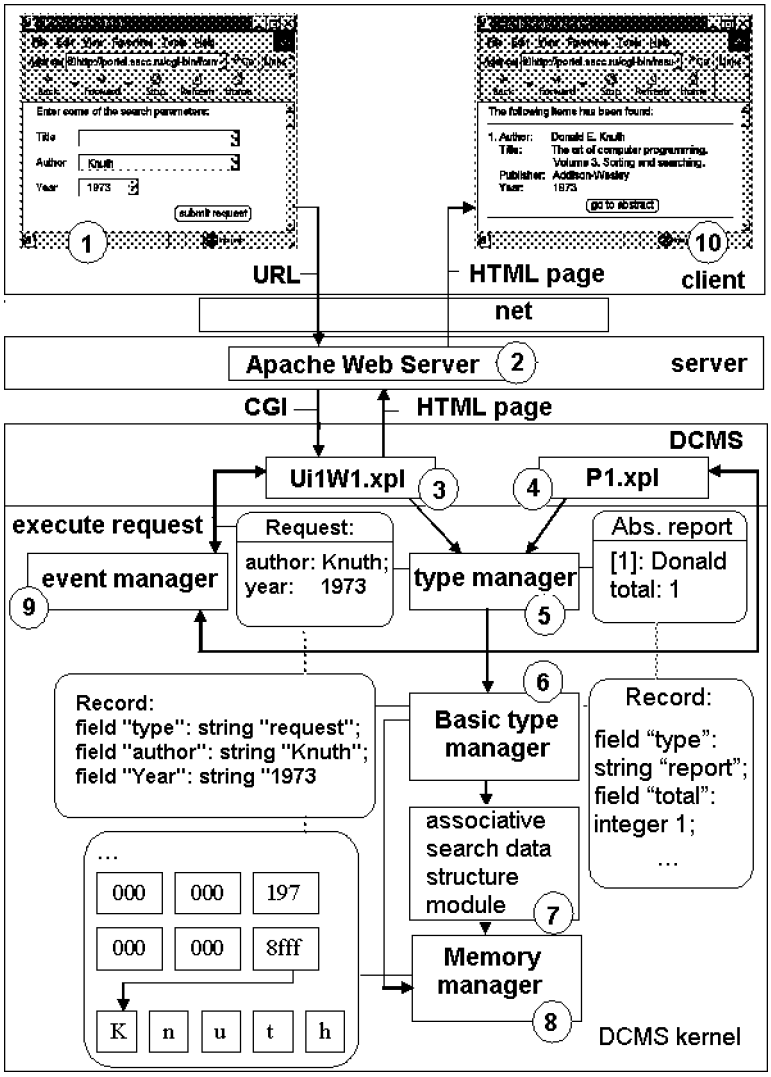

Figure 2

Let the user request information about books from the portal. He would like to narrow the search by specifying the author's name and the year of publication. He downloads the book search request form (1) and fills out the author and year fields in it. After he presses the submit button, the browser sends the request to Web server as an URL string:

```
http://portal.sscc.ru/cgi-bin/searchbook.elf?author=Knuth+year=1973.
```

The Web server receives the string, launches searchbook.elf and passes the author and the respective year. The Ui1W1.xpl is the first module triggered in the application. It converts the request from the Web server format

into the inner DCMS format. The Ui1W1.xpl uses a type manager (5) to construct a "request" and assigns the authors and the year properties.

The type manager itself uses the basic type manager (6) for an object allocation and creation of subobjects for the properties: it creates four objects of the void type (one for the request object and three for the properties kept in it as fields); it assigns the string value "request" to the second value, the string value "Knuth" to the third value and the integer value "1973" to the fourth value; and it inserts these objects into the first one as fields with the following names: "type", "author", "year". While performing these actions, the basic type manager uses the associative search data structure module (7) in order that form the data structure, which associates field names with their contents be formed. Both the basic type manager and the associative search data structure module use the dynamic memory manager (8) for the memory blocks allocation when creating data objects and associative data structures.

After the "request" object construction Ui1W1.xpl generates "execute request" event using the event manager (9). This initiates all the handlers of this event. The main work is done by the request processing module within the portal kernel P1.xpl (4). The request processor gets a "request" object as parameter and uses databases to retrieve all the "book" objects that meet the criteria of the search. Then the request processor constructs the "abstract report" object that contains all the objects found in the form of a sorted list. The "abstract report" object is passed to the Ui1W1.xpl module as a result of the "execute request" processing. The Ui1W1.xpl generates the HTML page by the abstract report and sends it to the Web server. The page contains a human readable list of the books found. The Web server sends this page to the user's browser (10).

7. Conclusion

Most of the DCMS kernel modules are implemented and used within the WinALT project. It was demonstrated that the kernel meets all the requirements that were expected. It is planned to finish the design of the kernel and to implement the whole of the interface description hierarchy as well as the development of standard system external modules with typical functionality for a wide set of applications. The DCMS is supposed to be used for the Web application construction, particularly, for portals and visualization of distributed computational processes.

References

[1] Ostapkevich M.B., Piskunov S.V. Basic constructions of models in WinALT // NCC Bulletin. Series: Computer Science. – Novosibirsk: NCC Publisher, 2001. – Issue 14. – P. 43–58.

[2] Ostapkevich M.B. The software development for imitational simulation of fine grain algorithms on clusters // Proc. Young Sci. Conf. – Novosibirsk, 2002. – P. 206–213 (in Russian).

[3] Filinov E. The selection and design of conceptual model for open system environment // Open Systems. – 1995. – Vol. 14, No. 6. – P. 32–46 (in Russian).

[4] Nepply, Platt. Programming Modula-2. – Moscow: Radio i Svjaz, 1989 (in Russian).

[5] Oberg J. COM+ Technology, Basics and Programming. – Moscow: Williams, 2000 (in Russian)

[6] Tsimbal A. CORBA Technology. – St-Petersburg: Piter, 2001 (in Russian).

[7] Ostapkevich M. Event-driven tools for open system design // NCC Bulletin. – Novosibirsk: NCC Publisher, 1999. – Special issue. – P. 15–22.

[8] Ostapkevich M. Expulsive tree data structures for fast data search by a key // NCC Bulletin. Series: Computer Science. – Novosibirsk: NCC Publisher, 1999. – Issue 10. – P. 73–82.

[9] Ostapkevich M. The open architecture of WinALT // Joint Bull. ICM&MG and IIS. Series: Computer Science. – Novosibirsk: NCC Publisher, 1998. – Issue 10. – P. 79–91.