Software Systematization in Critical Infrastructures

Leonid V. Arshinskiy
Irkutsk State Transport University
Irkutsk, Russia
larsh@mail.ru

Vadim L. Arshinskiy
Irkutsk National Research Technical University
Irkutsk, Russia
arshinskyv@mail.ru

Sergey V. Bakhvalov
Irkutsk National Research Technical University
Irkutsk, Russia
bsv@istu.edu

***Abstract*— The report is devoted to the problem of systematization of software in information computing systems for critical infrastructures. The basis of systematization is the principle of necessity. Necessity is considered in three aspects: existential (necessity for existence), functional (necessity for functioning) and goal (necessity for achieving the goal). This view allows us to introduce the concept of software layers, as well as a more rigorous approach to its classification.**

***Keywords*— critical infrastructure, software, classification, systematization, necessity.**

I. Introduction

One of the most important components of modern critical infrastructure is software. Today, software integrated into management systems and becomes an integral part of such systems. The creation of appropriate software engaged in teams of developers around the world. However, no computer program exists on its own, in isolation from the rest of the software and related hardware. This makes us consider the computer program as part of a large information-calculating system (ICS). Especially if it refers to the network of the ICS. Leaving aside the development of special software, focus on the study of its place and role in the ICS as a whole.

To study this issue, the first thing to do is to systematize software objects, find the place of the corresponding software in the ICS, to classify it. By dividing the variety of objects under study into classes, systematizing them, it is possible to bring order to seemingly random objects and phenomena, it is possible to develop methods applicable to a given class or group of classes, to order objects in a particular respect, to identify genetic relationships, and so on. The importance and usefulness of classification does not raise questions. A classic example is the biological classification.

In the field of software, classification is also present. However, the first acquaintance with it shows that there is no single approach. Some authors divide software into system, applied and instrumental, with further division into smaller classes [1-5]. Others divide computer programs into basic and special. They are combining system and instrumental software, as well as service programs into basic and application programs (AP) into special software [5-9]. The third group of authors speak about basic, system, service and application software [10, 11]. And so on.

Software is classify on purpose, platform, way of interacting with hardware, method of distribution, portability, etc. Variety of classifications. Both the methods of classification and the names of classes often differ from author to author. The same materials can be presented several views on the classification [1, 5].

Both taxonomy and hierarchy looks different in different authors. In some cases, the software is divided, for example, into system, application and special software. In one case division is flat – the whole set of computer programs are divided as a homogeneous whole [7, 12]. In other case some hierarchy is established [5, 12, 13]. For example, according to the degree of proximity to the hardware [5, 10, 14].

Many sources deal with the two-membered division: system and application software, with further deepening in each of these parts [7-9, 12-15].

It seems that many authors come from intuitive ideas about the classification of software objects or from established traditions. Sometimes the basic principles of classification are violated: one element is one class, the principle of completeness of division (proportionality), etc. [16]. It considerably complicates the issue. Not trying to find out the correctness of certain views, let's try to study it from, relatively speaking, formal-logical positions.

II. Necessity and sufficiency

The basis of the classification is not the problem solved by a particular program, and its need to solve these problems, as well as the mutual need between the programs within the information computing system (ICS) as a whole.

While talking about certain objects, including software objects, we distinguish three aspects of their existence:

- basic (existential) – object exists;

- functional – the object (existing) is functioning, at least partially workable;
- goal – the object (functioning) achieves its goals

Any objects can be considered from these positions, as well as, members of such a complex entity as the information – computational system (ICS): hardware, software, users, staff, developers, documentation, etc. While objects can enter into relations of necessity and sufficiency.

Definition 1. We will say that object A is necessary for object B, if the existence, functioning or achievement of the goal by object B is impossible without object A, or the loss of object A or part of its functionality means at least partial damage to B.

In this case, we also say that object A *has value* for B. We will write the relation of necessity as:

$$\neg A \to \neg B.$$

Necessity may exist in any of the three above meanings or a combination thereof. For example, the existence of the hardware is necessary for the functioning and goal achievement of any software product. Where it is important, the different types of need are denoted as $\to_e$, $\to_f$, $\to_g$, respectively.

Note that between the aspects of the existence of objects there is also a relation of necessity: (i) is necessary for (ii); (ii) is necessary for (iii). That is significant.

Definition 2. We will say that object A is sufficient for object B if the existence, functioning or achievement of the goal by object A also means the existence, functioning or achievement of the goal (at least partially) by object B.

The nature of sufficiency can be specified separately.

In the case of sufficiency, we also say that object A *is useful* for B. We write this as:

$$A \to B.$$

For example, the use of any of the text editor is sufficient to create a text file by the user. The appropriate program is useful for the user

One should note that in general the value of A for B does not mean its usefulness and vice versa. For example, if there are two functionally equivalent programs A_1 and A_2 in the system, then each separately has not the value, since the removal, reduction or loss of performance, or not achieving the goal by the program A_1 while maintaining A_2 (and vice versa) will not affect B. At the same time, each of them may be useful.

Similarly, if A_1 and A_2 are needed for B only together, the usefulness of each of them separately is missing.

Consider how these concepts work when classifying software for critical infrastructures and so on.

III. Software systematization

The basis of the functioning of the computer is the hardware, which we take into account. We also take into account the user. These are the two extreme points of the proposed systematization. All objects involved in this process are divided into layers. Hardware refers to layer (stratum) S_0.

The existence of the hardware is a necessary condition for the existence and functioning of the first layer of software, which is denoted by S_1. We are talking about programs that provide the operation of peripheral devices, input-output devices, programs necessary for the operation of the core of the ICS. For example, the PC motherboard with installed processor, memory and other major architectural components of the computer. The loss of any of these programs will result in partial or complete failure of the ICS. These programs are created by the developers of the respective hardware and are often unique to each device. In this case, the hardware (physically) can exist without software objects from S_1, but S1 is impossible outside the hardware.

The relation of necessity between S_1 and S_0 is mutual. Hardware is existentially necessary for S_1, S_1 is functionally necessary for hardware. Given that existential necessity is more fundamental, we believe that the relation of necessity between S_0 and S_1 is as follows:

$$\neg S_0 \to \neg S_1$$

At the same time, the software objects of the S_1 layer are necessary to controlling individual hardware components.

The next layer is S_2, which requires layer S_1. The software of this layer is used to control the hardware complex as a system. It can include operating systems (OS) in the traditional sense (UNIX, DOS, Windows, Linux, etc.). OS is not necessary software for particular hardware components but without it is impossible to link the hardware into a single system. OS is necessary to computer control as a system. Functionally, the OS is based on the S_1 for working with memory, processor, and other components of the hardware. In some case, the OS can work with the hardware bypassing S_1, however, completely OS from this layer cannot give up. Setting the rule:

If some object O requires layer objects $i_1,\dots\ i_k$, where

$$i_1 < \dots < i_k\text{, than } O \text{ belongs to layer } i_{k+1}.$$

Therefore, the OS refer to layer S_2.

Next, consider the application and instrumental programs (AP and IP). Someone refer them to different groups, someone unites them together. From the point of view of the solved problems, these are different classes of program objects. Let's look at them in the light of the discussed approach.

A necessary condition for the operation of IP appears OS, i.e. layer S_2. From this position, IP belongs to the S_3 layer.

AP, which is primarily understood here as software objects for solving application problems in critical infrastructures, is also unable to function without OS. At the same time, IP is not functionally necessary for it. However, if the purpose of AP existence is to provide high-quality information and computing services, then IP is necessary for AP in the goal sense (the quality of AP written with the help of tools, we believe above written without its help). That is, there is the third type of necessity - the goal one. It gives reason to place the AP in the S_4 layer.

Note that OS can also be developed using IP. I. e. in addition to functional necessity:

$$\neg\mathrm{OS} \rightarrow_f \neg\mathrm{IP},$$

there is a relation of target necessity:

$$\neg\mathrm{IP} \rightarrow_g \neg\mathrm{OS}.$$

Functional necessity has higher priority, therefore, IP is followed to OS, i.e. belongs to layer S_3.

There is another group of software objects called interpreters. They can also be considered as a kind of IP. But unlike IP including compilers, IP in the form of interpreters is functionally necessary for AP operation (whereas for the compiled software object the appropriate IP is not needed). Interpretable software works only if there is an interpreter. If the interpretable type IP (IIP) obtained using the compiling type IP (CIP), we can write the ratio:

$$\neg\mathrm{CIP} \rightarrow_g \neg\mathrm{IIP}.$$

In the same layer there is AP of compiled type (CAP):

$$\neg\mathrm{CIP} \rightarrow_g \neg\mathrm{CAP}.$$

That is, IIP is placed into one layer with AP, which created by means of compilation process (CAP) – S_4. Whereas, the AP of interpreted type (IAP) is placed in layer S_5:

$$\neg\mathrm{IIP} \rightarrow_f \neg\mathrm{IAP}.$$

It leads to the conclusion that although CAP and IAP perform the same task – to meet the needs of the user in information and computing services, structurally they belong to different layers. To the layer S_5, in particular, are macros running in some office packages, scripts, and other analogs.

The ending component of this hierarchy is the user. He needs to work with software objects of any of these layers. The largest of them is the layer S_5. Based on the above rule, we conclude that the user, as an object, refers to the layer S_6.

In case of a more detailed division by layers, the user is assigned to the layer S_{N+1}, where N is the number of the highest layer of software objects.

The principle of necessity also can be applied to classify software according to the tasks to be solved. Program objects are often classified according to their capabilities: what a program is able to do. This shows the principle of sufficiency. If the software system is able to solve any problem, it can be included in the appropriate class. For example, expert system (ES), which manages critical infrastructures, may be part of a decision support system and part of process control system. Some useful utilities, which contain malicious functions, may be part of service software class and malicious software class. And so on. If we base on the principle of sufficiency, the same program can fall into more than one class.

Application software is developed to solve user problems, which most often include:

- performance of professional functions;
- leisure;
- maintenance of computers and improving the convenience of working on the computer (utilities);

We should also mention the malware; group (iv); where the users are its developers.

As already mentioned, in the systematization on the principle of sufficiency classification is blurred. The same object can fall into different classes. Classification by necessity seems to be more consistent. For example, if software is necessary for solve user professional problems, it belongs to group (i). If it necessary for relaxation and leisure activities outside of their professional duties – to the group (ii). If without it decreases quality and convenience of work on the computer - to group (i). Finally, if it is necessary for unauthorized intervention in the work of computers - to group (iv).

The principle of necessity lays the foundation in software classification those functions for which the software is necessary for the user. Then a table processor is a tool for the spreadsheet, the utility with embedded malicious functions – malware (the primary user – unauthorized opportunities receiver). And so on. It allows to clarify the systematics in each layer. To do this, you should separate the main and secondary functions of the software, performing the classification by the main (necessary) tasks. Classification by sufficient (what the program can do) blurs the boundaries of the classes. More formally, if the relationship is declared:

$$O_1 \rightarrow F\&F_1;$$

$$O_2 \rightarrow F\&F_2;$$

where O_i are software objects and F, F_i are functionality implemented by objects, and

$$\neg O_1 \rightarrow \neg F;$$

$$\neg O_2 \rightarrow \neg F;$$

then objects O_1, O_2 are declared to belong to the class associated with the functionality F, where F, F_1, F_2 are the capabilities implemented by the program, and F is the capability that the user needs.

Typically, application programs are independent of each other and can be include to the same level of hierarchy. However if this relationship occurs between a pair of programs, a local hierarchy can be introduced for them. We believe that program *A* and program *B*, established in a particular ICS, are at the same level of classification, if none of them is necessary for the other, or they are mutually valuable to each other.

A few words should be said about the current trend of using virtual machines. A separate virtual machine is an independent object in which there are the same layers of the objects under consideration as in the real, but only in the "guest" version. However, in an extended context involving real hardware, the host and the hypervisor, the hypervisor occupies layer S_4 (we believe it was developed using the S_3 tool layer). It means that the layers of the "guest" machine, including the virtual version of layer S_0, should be numbered as S_{4+1}, S_{4+2}, and so on.

IV. Network software

We should also discuss the systematization of the network software. Its peculiarity, as well as the peculiarity of the network equipment, is that the "zero layer" here are the local computers united by the network. They are a necessary element of the entire infrastructure. Then the layer S_1 is the network equipment, which, in turn, can be viewed as a necessary condition for the functioning of the layer S_2-software objects that directly control the network hardware. The next level – S3 includes the network OS if it is installed on top of the local one, or the corresponding components of the local OS when it has network elements. Layer S_4 is software for realization network interfaces and protocols. Layer S_5 is formed by network applications, including network utilities, open source software and other programs running through the network, browser programs. Finally, in layer S_6, placed the software, which requires browser programs, which can also be used when working with critical infrastructures. The user in such a stratification is attributed to the layer S_7.

Application network software, as in the case of isolated computers, divided into classes according to the principle of necessity from the user's point of view.

V. Conclusion

The paper presents a possible approach to the systematization (including classification) of modern software. Most authors in the systematization implicitly proceed from the principle of sufficiency: the program is enrolled in a particular class based on what tasks it is able to solve. This blurs class boundaries because the same program object sometimes corresponds to different tasks. The paper proposes to proceed from the principle of necessity: the program belongs to the class, what tasks it must solve. Then the secondary capabilities of the software recede into the background and are not taken into account in the classification. The relationship of necessity between the program objects themselves is also taken into account. It allows systematizing program objects, "stratifying" on the principle of mutual need. Moreover, different types of necessity are introduced: existential, functional and goal, with a hierarchy between them. The latter allows you to keep the hierarchy of necessity, when program objects are mutually necessary for each other, but in a different sense. As a result, a hierarchy is built inside the software similar to the known: hardware, system software, application software, but with its own peculiarities. Accordingly, the applied software is stratified. In general, the principle of necessity should be considered as an approach to classification, which allows making a certain order in systematization of software and its division into classes.

References

[1] Alekseev E.G., Bogatyrev S.D. *Informatika. Mul'timedijnyj jelektronnyj uchebnik* [*Informatics. Multimedia electronic textbook*]. – http://inf.e-alekseev.ru/text/Klassif_po.html. (in Russian)

[2] *Klassifikacija programmnogo obespechenija* [*Software classification*]. http://mirznanii.com/a/308713/ klassifikatsiya-programmnogo-obespecheniya-2. (in Russian)

[3] Programmnoe obespechenie PC. Klassifikacija PO [PC software. Software classification] – http://mylektsii.ru/13-66978.html. (in Russian)

[4] *Klassifikacija programmnogo obespechenija po naznacheniju* [*Software classification by purpose*]. – https://wd-x.ru/klassifikaciya-programmnogo-obespecheniya-po-naznacheniyu. (in Russian)

[5] *Klassifikacija programmnogo obespechenija* [*Software classification*]. – http://www.intuit.ru/studies/ courses/3632/874/lecture/ 14291. (in Russian)

[6] Klassifikacija programmnogo obespechenija [Software classification]. – https://pandia.ru/text/78/325/2652.php. (in Russian)

[7] *Software Engineering. Classification of Software*. – https://www.geeksforgeeks.org/software-engineering-classification-software.

[8] *FREE online courses on the Basics of a Computer - WHAT IS SOFTWARE* - http://www.openlearningworld.com/books/Basics_of_a_Computer/WHAT IS SOFTWARE/Basic classification of software.html

[9] *What are the classification of computer software and definition?* – http://www.answers.com/Q/What_are_the_classification_of_computer_software_and_definition.

[10] Zhuravlev G. G., Kuzhevskaja I.V., Volkova M.A. Prikladnye pakety programm v meteorologii: jelektronnoe uchebnoe posobie [Application packages in meteorology: electronic training manual] (Federal'noe agentstvo po obrazovaniju, Gos. obrazovatel'noe uchrezhdenie vyssh. prof. obrazovanija «Tomskij gos. un-t». - Tomsk : TGU, 2006. - 1 jelektron. opt. disk). (in Russian)

[11] Klassifikacija programmnogo obespechenija. Vidy programmnogo obespechenija i ih harakteristiki [Classification of software. Types of software and their characteristics]. – https://www.altstu.ru/media/f/Tema-6-Klassifikaciya-PO.pdf. (in Russian)

[12] Thakur D. *What is software? Characteristics and Classification of Software*. – http://ecomputernotes.com/software-engineering/characteristics-and-classification-of-software.

[13] Maclnnes M. *Classification of software.* – https://prezi.com/dbxyq-gz1dle/classification-of-software.

[14] *Computer software.* – http://kcsecomputernotes.blogspot.com/2013/08/download-free-latest-unlimited-kcse.html

[15] *Various Classification of Computer Software*. – http://bankofinfo.com/classification-of-computer-software.

[16] Grjadovoj D.I. Logika: strukturirovannyj uchebnik (dlja vuzov) [Logic: structured textbook (for universities)] (M.: JuNITI-DIANA, 2003) (in Russian)