The Ministry of Science and Higher Education of the Russian Federation

PETER THE GREAT
ST. PETERSBURG POLYTECHNIC UNIVERSITY

V. E. Kovalevsky   V. A. Onufriev

INTELLIGENT CONTROL SYSTEMS ENGINEERING

Study guide

POLYTECH-PRESS
Peter the Great
St.Petersburg Polytechnic
University

Saint Petersburg
2021

P e e r r e v i e w e d b y:
Doctor of Technical Sciences, Chief Researcher
of St. Petersburg Federal Research Center of the Russian Academy of Sciences
S. V. Kuleshov
PhD in engineering sciences, Senior Researcher
of Saint Petersburg branch office JSC “Strela”
A. V. Kvasnov

Kovalevsky V. E. **Intelligent control systems engineering** : study guide / V. E. Kovalevsky, V. A. Onufriev. – SPb. : POLYTECH-PRESS, 2021. – 120 p.

Most of the focus of this guide is given to knowledge-based systems and their use, as well as to knowledge engineering as a means of working with knowledge that should form the basis of intelligent systems.

This guide can be used as part of the studied courses as well as when conducting research work.

This study guide is intended for students studying the courses “Knowledge engineering and knowledge management” and “Software development technologies”, which is included in the curricula for the educational program 09.04.01_17 “Intelligent systems”.

Printed by the Publishing Council
of the Peter the Great St. Petersburg Polytechnic University Academic Council.

ISBN 978-5-7422-7204-5

CONTENTS

INTRODUCTION

Nowadays, the trend of intellectualization has become very popular. Many consumers and manufacturers have considered modernizing their way of life and/or production by introducing some kind of intelligent systems.

However, the term “intelligent” should be used with great care and responsibility since not everything labeled intelligent is in fact so. After all, as with everything else, a simplification of concepts and a blurring of their boundaries occurs. Any neural network tends to be called intelligent, along with any digital sensor, any device included in a smart house (and the control system of a smart house itself).

However, when taking a closer look, it is easily can be seen that in most cases the use of the term “intelligent” is unfounded and not well considered, since this concept is inextricably linked to working in conditions of uncertainty, where there is no ready for use (pre-determined) algorithm, no complete data about the situation, and no clear understanding of the results to which one or another possible actions will lead. If, during its work, a system does not perform unforeseen experiments in advance, thereby generating new experience, if the obtained experience is not analyzed to identify new patterns, if missing information is not requested, then this system has nothing to do with intelligence.

Looking at the said above, it is easy to understand how difficult and science-intensive the way of design and development of any intelligent system. This is a multistep process in which various mathematical, logical, and programming tools are used.

Some of these are discussed within this guide. A large part of it is devoted to knowledge bases and, accordingly, knowledge engineering as the set of tools for carrying out various operations with them.

The first part discusses the concept of knowledge itself and the most important ways of representing it, as well as some matters of its obtaining (extraction, acquisition, formation).

The second part highlights the task of structuring knowledge, this is often a necessary step for converting knowledge into code. Here also shown such kind of structuring as representation of knowledge in the form of rules written in different logic languages: propositional, first-order and description. The implementation of manual and program structuring is shown using projects like Wikipedia and DBpedia as example.

Then the third part deals with the software implementation of knowledge bases by their formalization, i.e., translation to the appropriate languages. This includes matching the logic languages (first-order and description) with the means of knowledge representation languages.

The concluding fourth part examines the concept of intelligent knowledge based systems – a complex software product which helps resolve uncertainties by use of number of heterogeneous components, which should, however, synchronize their work.

When reading this study guide, it is recommended to view all materials as a tool for the following software implementation of a knowledge-based system in the form of a software product. This is especially relevant for materials not directly related to formalizing knowledge.

PART 1. KNOWLEDGE AND ITS PRELIMINARY PREPARATION

Intelligence and rationality

The term “artificial intelligence” became so common not only in field of computer technology and automation, but also in everyday life that no one even thinks about it anymore. But intelligence is not the same as rationality.

Rationality is defined here as the ability to perform logical operations (conclusions, generalizations, etc.) according to the basic laws of formal logic, i.e., it is the ability to think.

Intelligence is defined here as the ability to solve problems in conditions of insufficient or uncertain source data, as well as the ability to solve unknown beforehand type of problems.

If we analyze the problems which are solved using “artificial intelligence” and study their setup and methods for solving, it becomes clear that we are talking about rationality. For example, problems of object recognition or problems of machine vision in general usually deal with previously known objects or objects with a predetermined set of characteristics.

The tasks of an “intelligent” search always as well have a clear, predetermined setup of the task and fixed methods of how to solve it, that sometimes come down to complex, but nevertheless predetermined algorithms.

The task of passing the Turing test is much closer to solving problems using intelligence if emphasis is placed not on a computer agent trying to be indistinguishable from a human (for example, making mistakes typical to the latter), but instead on the ability to discuss any previously undetermined topic whilst demonstrating its understanding at an analytical and synthesis level, and processing data of a previously unknown structure.

Only a quick reflection on the difference between this problem and the previous ones is required in order to understand how far it surpasses the rest in complexity. It is even more complex in its principle solution: it is easy to make a scheme for solving the problem of object recognition (<discover an unidentified object>–<request a list of necessary features sufficient for recognition>–<sequently compare the features in the base with the features of the object>–<assess the level of overlap, determine the most suitable ones>–<draw a conclusion>), while even having a complete list of possible questions and possible answers on any topic (which is doubtful on its own) will nevertheless in no way contribute to the "ability" of the machine to analyze and synthesize random information.

Another example of solving problem using intelligence is the task of determining the intelligence quotient, for example, the Eysenck test (determining the level of IQ). We recommend anyone who has never taken such a test to do so, keeping in mind the task of creating a software agent which could gather the maximum number of points in the results, without prior knowledge of these problems.

Furthermore, it is important to observe your own process of solving these type of problems. How do you do it, and what do you use for it? For *any* question of these types of tests, you can formulate a logical answer, which will be the result of logical operations, i.e., thought process. Therefore, this process should be examined in more detail.

As an example let's take a problem that is common in such tests (Fig. 1).

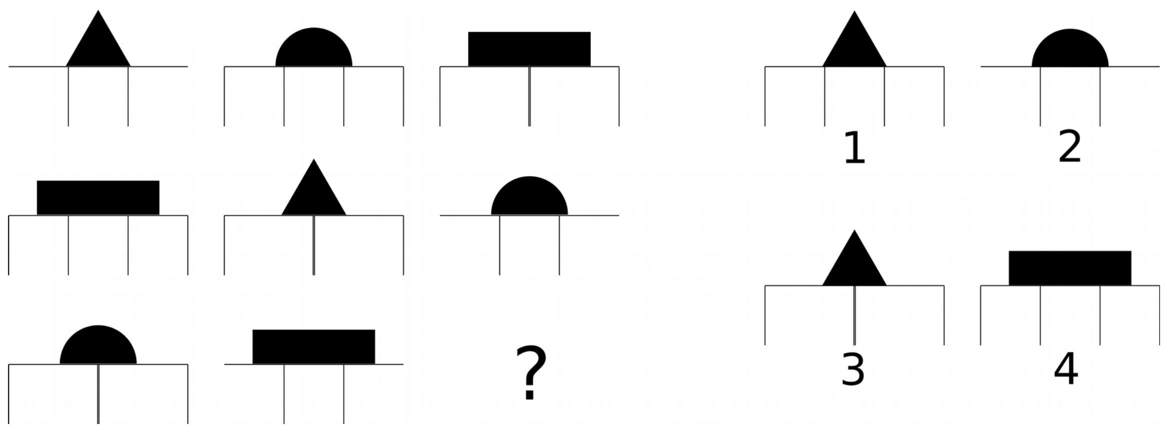


Fig. 1. Example problem.

The first thing which can be pointed out in solving it is the type of problem. Here we are dealing with a problem about a missing element. What gave this away? The presence of a question mark in the body of the problem.

So, our way of thinking here starts with an assertion

"?" indicates a problem of absence

Assertion – a completed thought in which one mentioned element is revealed through another. In other words, it must contain a subject and a predicate. In our example, the subject is "?", the predicate – "indicates", and the object – "problem of absence".

It should be pointed out that assertion itself, and not any thought, is the elementary unit of thinking – the process of logical conclusion.

What is next?

If you already have experience solving problems such as this, you understand that they can be graphic, mathematical, linguistic, and more.

Here we have a graphic problem, where the solution depends on the arrangement of pictures (we will skip the steps of reasoning how we classified it as such). Our assertion is

"The considered problem is graphic".

Furthermore, it is well known that

Any graphic problem has repeating elements or their repeating properties, which are necessary to define.

The last statement is not only an assertion but also a **rule** due to its universality shown by the universal quantifier "any".

Now, by combining the assertion "The considered problem is graphic" with the rule "Any graphic problem has repeating elements or their repeating properties", it can be concluded that

The considered problem has repeating elements or their repeating properties, which are necessary to define.

Thus, from the rule (about graphic problems) and the assertion (about the specific problem belonging to the graphic class), the logical conclusion can be made that this is where we need to look.

After several steps of such reasoning, we move on to solving it. What is needed for this? Knowledge of the rules (assertions with an element of universality) and the ability to apply these rules to the specific situation for which it was necessary, for example, to be able to recognize the repeatability of an element in a figure.

Could this problem have been solved without observing the rule defined above? Remember the first time you solved this kind of problem, not yet knowing about repeatability? What other rule did you know? The rule that these kinds of problems definitely contain a certain pattern. Furthermore, you also had in mind an implicit list of possible patterns, so after seeing them, you would recognize them immediately:

- repeatability of elements;
- mathematical patterns (continuous growth, changing numbers by law…)
- linguistic patterns – repetition of letters, words, change of letters by rule, for instance: a, c, e, g, i… and others.

As soon as you would see any of these, you would have tried in some way to apply it to this problem and, in the end, you would have discovered that in order to repeat the pattern, a specific element should be placed instead of the "?", and in order to do this, you would have needed to presume that the solution to the problem is to maintain the pattern.

This brings us to a conclusion that at the basis of any human solution of any kind of problem there is either knowledge of specific rules or the ability to guess and test these rules (which already speaks to a person having rules for checking and disproving rules). **There is nothing that shows a fundamental impossibility of giving these features to software agents**.

To sum up let's emphasize that

- knowledge of rules,
- the ability to make assertions,
- the ability to apply the rules to assertions

lie at the basis of making decisions in problems requiring the use of intelligence. However, they are not required in problems not relating to intelligence, for example, for image recognition (at least, explicitly). The problems concerning the use of elements of artificial intelligence and those not concerning it should be clearly divided since in order to solve them, different approaches and tools are required.

Thus, when designing intelligent systems, our task will be the implementation of three above mentioned possibilities in software agents.

However, before this, another important terminological point should be understood.

Information, data and knowledge

Now to continue our work we need to clearly understand its object and this requires understanding of such concepts as information, data and knowledge.

Before that, we should understand the process of defining concepts itself. It is often required to reproduce definitions, and therefore it is necessary to have a clear view of how this is done correctly.

There are several ways of defining concepts. We will use, mainly, one of them: **generic term (genus) and differentia** (distinguishing characteristic)**.** This kind of definition is called "**intension**". When using this kind of method, the closest generic term and minimum set of features distinguishing this term from others of the same kind should be identified.

A generic term denotes a more general concept than the one being defined. For example, the closest generic term of the concept "table" can be called "furniture", for "cat" – "mammal", and so on. The process of moving from a concept to its generic term is called **generalization**.

A differentia can be identified by more than one way, therefore the resulting definition can be different from that in the dictionary, but this in itself does not signify its fallacy.

Next, the terms "information", "data" and "knowledge" need to be defined since further on we are going to deal with knowledge, so we need a clear understanding of what it is.

Which of the three terms is the most general, i.e., includes all the rest? This would be **information** – any facts which can be perceived and processed by a person or computer. Obviously, knowledge and data are included in this term. Pay attention to how difficult it is to propose a generic term for "information". Due to this, this term can only be "grasped".

The definition of information given above is not a definition through a generic term. On the contrary, this definition is given in the form of examples (more defined concepts included in it). This is equivalent to a definition such as "a bird is a sparrow, goose, ostrich". These types of definitions are called **extension**, and they only make sense for such large concepts as "information" when an intension cannot be given.

It should be noted that the definition of the concept *information* says nothing about the way in which the information is presented. This allows us to distinguish the concepts *information* and *data*: **data** is information presented in a structured form. Thus, it can be presented in the form of graphs, tables, text, sound, etc.; the most important thing is the presence of order, clearly formulated and formalized. The fact that it is stored on a computer cannot serve as the criterion that we are dealing with data.

Knowledge is the least broad concept of the three. It must be structured and therefore it is data. However, not all data can be called knowledge. **Knowledge** is data presented in an understandable way for a human or computer agent about how to use, apply, and process other data. In other words, it is data about the ways of working with data, or metadata.

Data and knowledge are fairly easy to distinguish: if we see a table with information about a change in the exchange rate or about the readings of a temperature sensor in a reaction then this is data. While information

if this first data, given in the form of a graph, forms a specific figure
("head and shoulders"),
then this speaks of a quick
drop in the exchange rate in the near future

is already knowledge.

At this stage, the question already arises: how can a machine apply this knowledge? After all, even a person with a clearly formulated rule can quite easy experience difficulties in applying it.

For this, it is necessary to understand what form knowledge can take, and according to what criteria it can be distinguished from other data. It is also important to determine what form it should take in order to be used by computer systems.

Neuroinformatics and knowledge engineering

In the middle of the 20th century when the idea of creating artificial intelligence came about and was put into motion, scientists came up with two paths for developing this area. These two “branches” differ in their basic assumptions, while sharing a common goal.

In the first case, researchers relied on the assertion that there is currently only one known bearer of intelligence – the human brain. Hence, in order to replicate its functionality using machines, its structure first needs to be replicated. It is quite easy to see that this is a way of neuroinformatics, in which artificial neural networks are built.

In the second case, researchers focused not on the structure of the human brain, but on the content of thought itself: the assumptions and rules, assertions and facts it contains. They viewed their task to be modeling the processes themselves in order for a machine to be able to replicate the intentional process of thought itself. Their retreat from biological structures was explained by the fact that humans have already managed to create mechanisms not existing in nature (for example, airplanes), which are capable, to some extent or another, of repeating or even exceeding their natural counterparts.

For several decades, both of these lines of development paralleled. However, each of them has its own drawbacks, which have not allowed them to adequately solve the increasingly complex problems which they are faced with.

Now let's look briefly at the fundamental principles of neural networks in order to get later better understanding of the differences between the two mentioned approaches for developing intelligent systems.

The most elementary element at the base of a neural network is an artificial neuron – a mathematical model of a biological neuron. This model has one or several input parameters (corresponding to short sprouts of a neuron – dendrites) and one output parameter (corresponding to a long sprout – axon). The signals reaching the neuron through dendrites are added up, specially processed by the neuron, and sent to the axon.

This means that the artificial neuron itself is nothing more than a mathematical dependency function of one output parameter on the sum of input parameters. This type of function is called an activation function.

When joined together, the neurons form a network. An artificial neural network, therefore, is represented in the form of a graph consisting of vertices (neurons) and edges (input and output processes).

The area of connection of two neurons (the "exit" of one with the "entrance" of the other) is called a synapse, and this connection regulates at a chemical level the gain of the conducted signal.

In mathematical interpretation, this means giving each of the graph edges its own weight, which implies that the signal running across is multiplied by the value of its weight, thus the signal arrives at the next neuron already weighted.

Let's look at Fig. 2 which shows part of an artificial neural network.

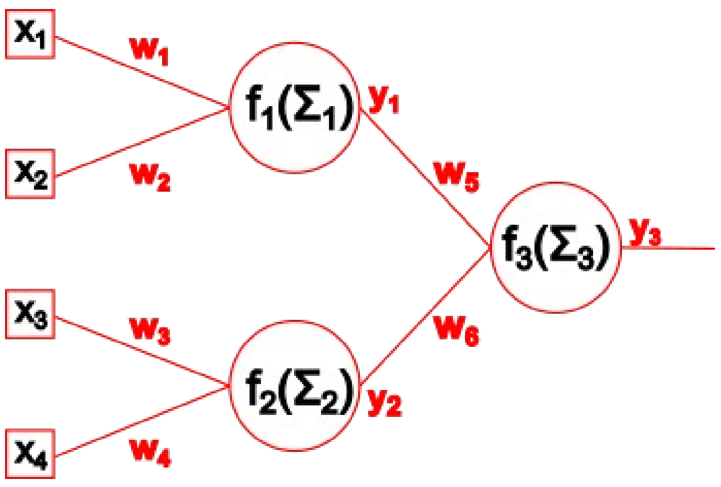


Fig. 2. Part of an artificial neural network.

Thus, the dependency of the neural network's output from its inputs is expressed in the following form:

$$y_3 = f_3(w_5 \cdot f_1(w_1 \cdot x_1 + w_2 \cdot x_2) + w_6 \cdot f_2(w_3 \cdot x_3 + w_4 \cdot x_4)) \qquad (1)$$

Further, it is usually said that a neural network should be trained, by changing its graph weights in such a way that any combination of signals at the inputs is processed correctly.

Thus, it turns out that all intelligence of a neural networks is based only on a set of all its weights, and since it can be said that neural networks are "able" to process data correctly and make conclusions based on it, then we can talk about the **knowledge** which the neural network possesses.

At the same time, this knowledge is presented in neural networks **implicitly**. This causes a complex, not always solvable, problem of transferring knowledge from a set of weight matrices into a form which can be interpreted by a human. For example: what knowledge can be extracted from expression (1) if the numerical values of all presented weights are known? After all, this is knowledge presented in neural network form.

Furthermore, change in the number of inputs and/or outputs requires retrain of the network, and the knowledge presented in it before becomes invalid.

Moreover, in order to train a truly complex neural network (when the number of its layers is measured in tens or even hundreds), there are significant requirements to computing equipment, which at one time noticeably influenced the enthusiasm and rate of research in this area.

This forces us, despite all the undeniable positive qualities of neural networks, to search for other methods of representing knowledge and move to the second path of the two presented above, where our task is to replicate not the form of the thinking apparatus, but its content.

Then we should move from examining neuroinformatics to a broader area – knowledge engineering.

Knowledge engineering – is a part of science about artificial intelligence, that researches issues related to knowledge, specifically its extraction, structuring and formalization. These are the three main tasks of this discipline.

As a result of completing these three tasks, knowledge is presented in an explicit form, "understandable" for software agents, since they are the final subject of knowledge application. The necessary forms for this we will discuss later. But now, we should concentrate on the difficulties which stand in the way of working with explicitly expressed knowledge.

The main way of storing knowledge is not as numbers, as in the case of neural networks, but as **symbols (text)**. This means that by default knowledge bases – specially organized data stores – do not have a built-in mathematical apparatus. This implies, by default, a lack of ability to approximate mathematical functions. In this regard the process of creating knowledge within such stores differs greatly from creating knowledge within a neural network (i.e., its training).

Knowledge bases can be filled in three ways: introducing already existing knowledge, by inference engine computing a deduction and obtaining by connecting various data analyzers.

In the first case, the already existing knowledge (facts, rules) are inserted directly into the base. In order to do this, the previously mentioned procedures of extracting, structuring and formalizing knowledge should be completed in sequence. It should be noted that this process is amenable to automation, but this task will be considered separately.

The second case is about using logic (for example, first-order logic, description logic). In this case new knowledge is obtained based on already existing one by applying the rules taken from the base to the facts contained within it. In other words, in simple cases we are dealing with **deductive inference** – a conclusion of concrete (particular) facts based on more general rules. This also includes the search for specific patterns in existing knowledge. This is a move from general to specific, for which various **inference engines** are used – programs which make deductive inferences of knowledge (facts and rules).

The third case is the most complex and knowledge-intensive and has two possibilities: conduct **inductive inference** (from specific to general) based on existing knowledge in the base or based on a random data set.

In order to carry out the first possibility, special software analyzers are needed, operating from the so-called "**open-world assumption**", which implies that everything which is not explicitly recorded in the knowledge base can, nevertheless, exist (for example, if the base does not contain information about the existence of immortal beings, their existence is still possible).

In order to analyze a random set of mathematical data, a use of individual modules (including, those of neural networks) is needed, since knowledge bases on their own "are not able" to independently extract mathematical patterns, since their own structure and logic of work is not strictly mathematical.

It should be highlighted that the implementation of possibilities of deductive and inductive inferences is a key to creating intelligent systems based on knowledge.

However, in order to carry this out, it is first necessary to initially fill the base with knowledge, based on which it is then possible to make inferences. And in order to move to this task, we should first look at the internal organization of the knowledge base itself and at the basic forms of its contents.

Classes, individuals, and properties

A knowledge base is also represented in the form of a graph with vertices and edges like neural network. However, unlike neural networks, a knowledge base is not a mathematical model with inputs and outputs. Its content is comprised of some elements presented in text form and the connections (relations) between them, that is also text.

When we speak about knowledge bases, we are speaking about knowledge represented by symbols. There are certain difficulties along this path which need to be somehow resolved.

The study of symbols and their meanings is addressed by the science of **semiotics**. We are only interested in certain aspects, which include the **triangle of reference (Frege's triangle)** shown below (Fig. 3), introduced almost 100 years ago, but still used today.

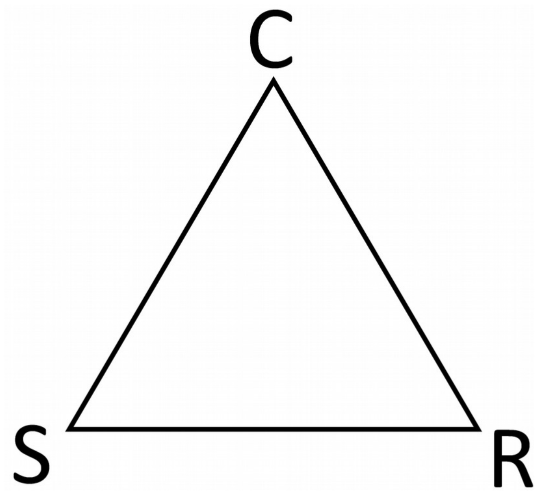


Fig. 3. Triangle of reference.

Study the image carefully: "C" means "concept", "S" – "symbol", and "R" – the referent (thing). Now we need to understand the connection between this construction and our further work.

The plain content of the knowledge base is represented by symbols. However, the task of the knowledge base is to describe the **concepts** and **instances (representatives) of these concepts** behind these symbols. In the triangle of reference, they take, respectively, the upper (concept) and right (the specific objects they refer to) parts.

The best analogy which can be given here is object oriented programming with its two very similar terms: **Class** and **Instance of Class (individual)**.

A knowledge base, besides the classes and their instances (individuals), must also contain **properties**, i.e., characteristics which can refer to classes and instances (more often the latter). The properties have names (e.g., "mass" or "presence required") and values (e.g., "500 kg" or "steel cable").

"Class" will be considered simultaneously in two aspects: as a concept (Fig. 3), determined by the aforementioned method through a generic term and difference of kind, and as a set (totality) of objects with a specific common collective feature.

In the first case, the class is defined through another, broader (more common) class. Thus, the class "Accidents" can be defined through the class "Events", while noting the features that distinguish the incursion of such situations from other events, for example, poor predictability, unwanted consequences, and so on. Then it is about establishing the relation of "**subclass of**". A class is defined as a subset (subclass) of a larger set (class).

In the second case, a class is determined through a set of features, which is satisfied by its individuals, for example: "Cranes with a lifting capacity from five tons and more". Of course, it is also possible here to define it using a generic term ("Crane"). However, when describing the class, the relation of "subclass" was not established clearly. Instead, a **restriction** for the property "lifting capacity" was introduced. Thus, the class is defined as a set of individuals satisfying a given restriction.

The **class instances** themselves (individuals) are defined exclusively by their belonging to one or several classes at the same time. For this, the relation **"type"** is used – one of the most widespread in a knowledge base graph.

Now it can be said that any vertex of this graph is either a class or an individual. A vertex has an identifier, a symbolic name, which corresponds to either a class or an instance, and this identifier, which is chosen by the creator of the knowledge base, should define it clearly. The edges of this graph are the relations between the vertices (e.g., the mentioned type, subclass, etc.).

The form of representing knowledge base in this way, using the described graph, is called a **semantic network**. It is not the only possible form (there are also frame systems and production rules), but it is the one which is looked at below.

In order to do this, we should look at the difference between semantic networks and other networks and introduce specific terminology.

Syntax, semantics, context

As it can already be guessed, creating a knowledge base, when talking about the process of filling it, comes down to describing its elements and the relations between them, i.e., to describing its graph.

At this moment directly creating these descriptions and these connections, we find ourselves at the layer of **syntax**. In terms of grammar, syntax (from Greek σύνταξις) is about principles of constructing and linking sentences in a specific language. In terms of formal languages (including programming languages), syntax is a set of rules for forming more complex expressions from symbols used as the basis. In terms of knowledge bases, we are interested in syntax as rules for describing and linking classes and instances.

In order to talk about the rules, it is necessary to define the sources of rules. In the case of knowledge bases, rules are defined by so-called **knowledge representation languages** (KRL), which, as in the case of programming languages, can cause syntax errors while checking the code.

It should be noted that the syntax layer is the easiest to check for errors, since syntax analysis is the process of comparing the given set of symbols with a well-defined beforehand template of a standard data structure.

However, what if it is necessary to check not only the knowledge representation form itself but its contents as well? At that moment, the syntax layer becomes insufficient.

Imagine that your task includes checking that the hyperlinks in a system of interconnected electronic documents (or pages) work properly. In this case, the work in the syntax layer involves two checks:

– is it really a hyperlink, does it redirect the user when clicked on;
– doesn’t it direct to a non-existent page, when following it leads to the error “404” (“page not found”).

However, it is clear that in order to check that the hyperlinks work properly, it is also necessary to be sure that the link with the text “industrial equipment” actually leads to a page with a description of industrial equipment and not of popular medical supplies. Checks of this kind have no obvious solution.

Here begins the **semantics** layer (Greek σεμαντικος) – a section of linguistics which studies the meaning of symbols, their notional (semantic) load. This lead us to the conclusion that, the above mentioned check of the link “industrial equipment” belongs exactly to the semantic layer.

Thus, in order to carry out the mentioned check of the link, a software agent should follow it, determine the title of the page, identify its semantic meaning, and correlate it with the semantic meaning of the hyperlink.

The considered form of a knowledge base is a set of semantic (notional) connections: class-class, class-instance, instance-instance. Building bases is not simply a process of inserting links (as in the Wikipedia project), but a semantic description, i.e., a clear indication of the nature of the connections between the elements, e.g., an instance belonging to a class (“a quadcopter is an unmanned aerial vehicle”) or a cause-and-effect relation between events (“violating the process of desalinating oil leads to a reduction in the quality of gasoline fractions”) and so on. This form of representing knowledge bases is called **semantic networks**.

The tasks of semantics also include issue of finding way by which the meaning of more complex concepts can be derived from the meanings of simpler concepts without human involvement, taking into account the syntax of links between them (for example defining something using a generic term and its differentiating features).

However, here the next more complex problem arises. Some words have multiple meanings, and some are homonyms. In both cases, one word has multiple meanings. This creates some difficulties, since, even if the full list of possible meanings of a specific concept is known in advance, how can it be determined which of them is used in the given situation?

This takes us from the semantic layer to the next layer – **context** (Latin *contextus*). Context is a broader semantic field about the relation between a specific element and its surrounding elements which in fact determine its interpretation.

Let’s take for example the concept “pot”. In the context of kitchen equipment, this concept means a cooking container usually made of metal. In the context of gardening and botany, the concept “pot” means a container (often clay) for planting flowers, small trees, etc.

Thus, it is the surroundings, or a context, which gives understanding which meaning of the concept should be used this time.

But it is worth remembering that any situation and any concept can be looked at in several contexts at the same time with varying degrees of generality, for example, the process of exchanging data between sender and receiver can be considered in a social context (with its rules and conditions), as well as in the more general context of the time and place of the contact.

However, the context of a message or concept in general may be unspecified or unknown. In this case, in order to choose an appropriate meaning for the concept, it is necessary to have an idea about the next layer after context – **pragmatics** (Greek πραγμα), which studies the goals and desires of the interacting agents, which determine the necessary contexts and, as a result, the meanings of the concepts. Put otherwise, pragmatics studies the connection of the subjects that use linguistic means with the contexts and semantics of these means.

Now, having examined the main aspects relating to knowledge, we will move on to setting tasks which can be solved through knowledge-based systems.

Tasks of knowledge-based systems

The very first task which is associated with using knowledge-based systems is the **information search** (both online and in any other sources).

The need for developing this area is seen when working with internet searches. Of course, such a problem did not exist at the beginning, when the Internet had just come about, when it had yet to offer surfing pages and global searches and only helped with looking up documents in stores and granting access to them. However, when the idea of Web 1.0, and later Web 2.0, emerged and was implemented, the problem of a keyword-based search in the search engines became clear.

The fact is that, by default, search programs do not understand the meanings of words and "perceive" any information on a page as nothing more than a combination of symbols with unknown content, an undefined context and which is used for an unknown purpose. What problems does this entail?

First of all, we are talking about showing results, many of which are not relevant to the query. The reason for this is that the context of the query is not taken into account, therefore occur problems with words of multiple meanings when the results include pages relating to completely irrelevant meanings (paintbrush instead of vegetation brush). This problem can be solved, for example, with knowledge of statistics, of more widespread contexts, which determine the appropriate meanings of the words.

Furthermore, search may suggest pages where the requested word simply appears and is not the subject of a text or page. In order to solve this problem, the search agent should be able to analyze the number and quality of the links (relations) between this word and the other words on the page.

On the other hand, a simple search by keywords does not give all results because it does not take into account, for example, searches by synonyms and other names (for example, when searching for information by the word "Jaguar" using standard tools, you will not see pages containing its scientific name *Pantera onca* instead of the searched word). This includes taking into account metaphorical expressions, common names, and other contextual definitions.

In order to solve this problem, a knowledge-based system uses an already structured knowledge base, in which concepts are stored taking into account the contexts, synonyms, and other features (this is one of the purposes for which semantic connections between elements are created).

It is worth special mention that a knowledge base can and, rather even should, be distributed, which means that a specific system module can access not only a local base but a remote base as well.

Unlike the task of extracting knowledge from already prepared bases, the task of **extracting knowledge from unstructured sources** is one of the most labour-intensive due to the need to first determine the context of the source material itself and then to conduct an analysis of the connections and meanings of the elements contained within the source wherein, the content and order they are presented in is not known in advance – and only afterwards solve the task of translating the extracted knowledge to the expected and predetermined form.

This task requires the use of a syntactic or other kind of analyser which creates materials for a knowledge base to store. Moreover, the system should constantly perform self-learning using existing and continuously gained new experience, adding the new knowledge (patterns, rules) to the base.

In addition, knowledge-based systems can perform a **teaching function**. Knowledge bases can be used to create training and testing programs, to create detailed analytics on the mastering of materials by topics, and to identify the rules and dependencies.

Next can be mentioned the task of **maintaining the data integrity** of systems. Integrity is understood as the coherence (consistency) of information. On the one hand, this means checking the materials for conflicting facts or rules, and, on the other hand, it is about the aforementioned syntactic and semantic integrity of references. With the increasing volume of materials, this task becomes all the more difficult for "manual control".

This can also include the task of **storing corporate knowledge**, issue that is relevant due to the direct dependency of a company's performance from the knowledge of its employees, which is always includes the "human factor" (specific worker can resign from the company before he shares his accumulated knowledge, and so on).

To the next task of a knowledge-based system can be given the general name **task of classification**. The process of solving it is different from a similar process that uses neural networks, where the input is usually raw data from the object being classified or the object itself (for example, a picture). Then the input is classified based on non-obvious calculations of the neural network. But in the case of knowledge-based system the input should be the features themselves in a specific, formalized form. Since mathematical operations (for example, processing a picture in order to extract the features) are not provided in a knowledge base, the neural networks (and other mathematical modules) must do a preliminary processing of the input data in order to solve this task (for example identifying on a picture the facts of the presence/lack of a certain kind of curves and/or their quantitative characteristics). This data is then used as the input of a knowledge-based system.

A more general case of the classification task is the task of **assessment**. Here, the methods or means of reaching the goal can be assessed in terms of its effectiveness; the level of certainty that some event or scenario will occur can be calculated; the result itself can be assessed for compliance with the initially set goals.

To this also relates the task of **predicting**, which includes, for example, choosing a more likely scenario of the further sequence of events based on the current indicators of the process, rules, and accumulated practical experience.

The task of **control in technical systems**, which involves maintaining of designated objective functions of control task at certain intervals, also requires interaction with other subsystems – for example, with a system that interconnects with the main control loop and its object, to which control actions are sent, that are chosen by the system based on technological parameters from sensors, history, and so on. Here the control signals can also be processed in advance using mathematical tools. Knowledge-based systems within the technical systems control are capable of solving uncertainties (will be explained in the fourth section of this guide).

In the case where a system is initially focused on working with the user, consulting and helping them with decision making, it can be assigned to the class of **expert systems**. At the same time, it should be noted here that not any consulting system can be called expert, since for this, the latter should have three mandatory components:

- the knowledge base itself;
- inference engine (*reasoner*);
- user interface with a subsystem of explanations.

In connection with this, any expert system can be asked the following questions:

- In which of the knowledge representation languages is the base implemented?
- According to what principle does the inference engine work, and how is it realized?
- How is the information displayed to the user transformed in order to take on a readable form?

However, at the heart of any knowledge-based system is a knowledge base. Therefore, in this book, we will look at, in turn, all the tasks and steps connected with preparing knowledge, the software implementation of its storage, and with its use.

Now let's study the types of knowledge and the forms of its representation in order to move further along in this area.

Classification of knowledge

Knowledge can appear in three main forms: facts, terminological axioms and role axioms.

Fact is an assertion which either describes the relation between two individuals or fixes belonging of an individual to a certain class. The relation *r* between the individuals *a* and *b* is denoted as *r <a,b>* or *arb*, and the belonging of individual *a* to class *C* as *a:C*. The totality of all the facts is called a **system of facts** and in international terms **ABox** (assertional box).

Terminological axiom is an assertion that fixes either the equivalency of two classes or the inclusion of one class in another. In the first case, the equivalency of classes C and D is denoted as $C \equiv D$, while in the second, the inclusion of class C in class D is denoted as $C \subseteq D$. The totality of terminological axioms is called a **terminology of concepts,** denoted as **TBox** (terminological box).

Role axiom is an assertion which indicates either a transitivity of some role or the equivalency of one role to another, or that it is a subrole thereof. In the first case, the transitivity of the role *isMadeOf* is denoted with the axiom Tr(*isMadeOf*) [Zolin, 2018]. The equivalency of the roles *canCause* and *canResult* is denoted as *canCause* ≡ *canResult*, similarly subrole is written as *products* ⊆ *results*. The totality of role axioms is denoted by **RBox** (relational box).

We will consider axioms in more detail later in another section of the guide.

Here, the meaning of the term "role" should be highlighted separately. In the assertion

roboticArm hasPart effector

the element "hasPart" can be called the role, property, predicate, relation. All of these terms, albeit similar, are nevertheless not equivalent: for example, the predicate is equivalent to the predicate in an assertion and is expressed by a property, but the property can act not only as the predicate but as the subject and object as well.

However, this concerns the classification of knowledge according to form and not according to content. Content-wise, knowledge can be classified, for example, according to its depth and source.

The first feature of classification is especially indicative, since it is an important characteristic for assessing the overall quality level of a knowledge base: does the knowledge base contain **deep** knowledge or does it only present **superficial** knowledge? The latter is synonymous with knowledge that is obvious or clear to any non-specialist, for example,

a cat is an animal

a neural network has neurons.

This kind of knowledge can be created in large amounts but expert experience and skills are not required to produce them. This means that in most cases they are not of value.

It is clear that the main part of any knowledge base should be created from deep knowledge. An example of such knowledge is, for instance, the assertion that the simultaneous combination of specific values in specific intervals obtained from specific sensors leads with a specific degree of certainty to a specific malfunction. Another example is the knowledge that, in order to improve specific maneuvers, a specific aircraft requires specific equipment with specific (well-defined) settings.

On the other hand, knowledge can be classified according to its source as **primary** or **secondary**. This classification is important in terms of storing knowledge: primary knowledge is obtained during direct interaction with the source, while secondary knowledge is obtained from analyzing literature describing the original source. In connection with this, knowledge obtained from an encyclopedia (including Wikipedia) is secondary, while from scientific articles and reports is primary.

Furthermore, topic-wise, knowledge can be divided into the following types.

What-knowledge describes the properties of an entity (instance, class or relation) and its belonging to a class (or its inclusion in a more general class), for example:

– tank 102 has a critical temperature of 94 degrees C,

– a neural network is a mathematical model.

Why-knowledge describes a cause-and-effect relation between individuals. It is especially worth highlighting that this type of knowledge describes independently (often uncontrollably) occurring consequences of some actions or events:

– an increased level of liquid in a tank triggers a sensor,
– the incorrect value of the checksum causes a delivery error.

What-for-knowledge, unlike the why-knowledge, always implies the intention of completing some action and describes its desired result:

– press button 6 for a smooth decrease in temperature,
– Start() function is needed to activate the transfer of cargo.

How-knowledge directly describes processes: their sequence, conditions for transitioning to new steps, interaction between active participants:

- to reduce the costs of production, 50% of resources should be invested in increasing productivity, 30% for reducing permanent expenses and 20% for reducing irregular expenses,
- two conditions need to be met in order to ship goods: receipt of payment and availability of goods in stock.

When-knowledge describes the sequence and time when some event or action takes place:

- first, the application is launched in 20 seconds, then each query is processed in no more than 1.5 seconds,
- after launching, each robot of the mobile group waits for two seconds for a query from the leader, otherwise protocol for determining the leader is initiated.

It is worth mentioning separately that, along with the semantic and logical models of knowledge that are examined in this study guide, there are frame and production models as well. About the latter it can be said that they are built by creating sets of rules in the form “if – then”. For these goals, various means can be used: from logical languages like “Prolog” to neural networks, which in this case should be developed from the beginning as a model of cause-and-effect connections between their input and output.

Now, having an idea of the classification and types of knowledge, we can move on to the first big task of knowledge engineering – the task of obtaining it.

Knowledge obtaining

The first question which should be asked of knowledge engineering is: what is the source of knowledge?

In order to answer this, it should be remembered that knowledge is specially structured data which can be used by machines, software agents, for their tasks. An example list of tasks is presented above. This kind of knowledge is stored in a limited set of forms – semantic knowledge bases, neural networks, text code in special languages and so on. They are, in fact, the final stores of knowledge, where knowledge is presented in a clear form for machines, in a **formalized** form.

However, since in reality knowledge is more often used by people and not machines, it is obvious that people did not require storing knowledge in this form until recently. In connection with this, all the knowledge accumulated by people is presented at best in a **structured** form, in other words, in the form of schematics, graphs with named connections, sequence diagrams and other forms.

However, in order to transform knowledge into any form, knowledge in any implicit form (at least in the form of text, outlines, notes) should be obtained from its original source, which is, unfortunately, for the most part is a human.

Knowledge elicitation

In the case of direct work with a human, we encounter the process of **knowledge elicitation** [Gavrilova, Kudryavtsev, Muromtsev, 2016], which means the process of a knowledge engineer (analyst) interacting with an expert (a human source of knowledge) with the goal of identifying and recording expert knowledge (their reasoning process when making decisions, the structure of their presentation of the subject area).

This process is particular in that it is difficult for the human alone to obtain and formulate the content of their own mind – sometimes impossible to do on one's own, due to the fact that, firstly, this process is uncustomary for practice-oriented people and therefore requires developed reflex skills, and, secondly, it is almost impossible for an expert to trace the whole chain of their own reasoning themselves: for them, some inferences and conclusions are so obvious that they do not realize the missed intermediate conclusions and presuppositions in their own reasoning. For example, the reasoning which appears obvious to an expert

instability of the magnetometer to interference -> replacement of the magnetometer is required

is of value only when revealing the intermediate steps:

instability of the magnetometer to interference -> high dispersion of readings -> inability to determine the direction of the robot -> replacement of the magnetometer is required.

Thus, the goal of a knowledge engineer is, from the one side, to identify and clarify all obvious steps in the reasoning of an expert.

From the other side, an expert can believe that they have less knowledge than in reality, and therefore it is necessary to also “retrieve” from their minds content which they themselves may not be aware of.

In most cases, knowledge elicitation involves the direct interaction of an analyst (group of analysts) with an expert (group of experts). Depending on the type of behaviour of the analyst, methods of communication are separated into **passive** and **active**.

Passive methods include observing the work of an expert, listening to his lectures, and so on. What is special about these methods is the minimal interference in the expert’s direct work and wasting of his time.

Active methods can also be divided into individual and group. The first type includes interviews and questionnaires, while the second includes organizing for experts “brainstorming sessions”, round tables and various foresight and role-playing games. This means that in both cases the analyst does not simply record material, but asks questions and heads the work of the expert group.

The forms of work can vary, but the focus should always remain constant: of special importance is deep knowledge, which can be represented content-wise as knowledge that describes, for example,

- the nature of how elements, indicators are connected. This can include formal dependencies, schematic/functional diagrams, knowledge of compatibility/inconsistency of certain properties or their meanings;
- cause-and-effect relations, which can include sequences/instructions (which actions are correct or incorrect for which goals in which conditions), rules of goal setting and rules of planning, property indicators (which external signs

indicate which internal properties are present, and vice versa), indicators of future events: signs for predicting the future;

- various classifications: hierarchy of classes (how association to one class means association to another), class features (individuals belonging to a certain class have a number of which features), rules of classification: the presence of certain properties implies association to a class with its features, reliability level of its classification signs, and its priority.

It is this knowledge which should make up the main content of the results from working with experts.

Knowledge acquisition

However, the process of obtaining knowledge from experts can be somewhat automated: part of the process of acquiring knowledge is passed on to machine agents.

This work names a different phenomenon: **knowledge acquisition** – the process of an expert filling a knowledge base using specialized software tools. The idea is that the control program itself suggests questions to the expert, stores and specially structures his answers.

These tasks, of course, should be done using not simply a knowledge base, but a **knowledge base management system** (KBMS) – a special software package which allows for the knowledge bases themselves to be filled and changed (including independently), while also implementing an interface for interaction between the base and user.

At the same time, the system can either take information from an expert in the form of previously completed templates, or by directly asking them questions and holding a dialog with them, each time taking into account the previous answers – in other words, implementing a certain strategy.

In the first case, for example, tabular forms are prepared for the expert, to which they can enter information on their own. So, one of the fields of the table can be "Possible forms of emergency situations" when producing nitrogen. Another field opposite each situation can contain its most common reason, and in the third – a node of the system which can be damaged, and so on. Next, KBMS uses this table for input, transforms all the information into code in one of the knowledge representation languages. In this case, the analyst compiling the templates should know clearly in advance what knowledge will be required from that specific expert.

Organizing a dialog system (second case) is required in situations where the analyst is poorly aware of this area of knowledge, when it is impossible to clearly articulate in advance questions for the expert or to compile a ready template. In this case, it can be relied on that any area is described by the three forms of knowledge already mentioned: facts, terminological axioms and role axioms. Meaningfully, it should result in assertions, the content of which will reflect the forms of deep knowledge given, for example, in the previous paragraph.

Thus, an example sequence of actions in this case can be the following:

1. The program asks an expert to list the basic, most important concepts used in their field. Later on, they will become the main classes (which in this context also have the sense of concepts).
2. The program requests that the introduced concepts be defined, denoting them as subclasses of more general (broader) concepts.
3. The program requests the features of the introduced classes to be specified.
4. The program asks that the rules of classification (assignment to the specified classes) be indicated.

Then we proceed to a system of facts.

5. The program requests critical examples of the subject area and asks that they be connected to the classes to which they belong. For example, indicating the main accidents discussed and described is an instance of the class Accidents.
6. The program asks for all of the most important **literal (datatype)** properties to be listed (i.e., existing values in the form of numbers or strings, and not in the form of other examples) for each example. For instance, the Wi-Fi module is characterized by a maximum level of consumed current, a maximum and

minimal range of connection between two points, and so on. Pay attention that this is about the properties themselves and not about their values.

7. The expert enters the values of the specified properties, the ones which are the most important to know, as well as the restrictions for possible values of the properties (their acceptable intervals, the number of possible values within the properties of one individual, and others).
8. Inputting the most important types of relations with other examples of the same or another class – such properties are called **object** properties. They should be entered for each instance.
9. Inputting restrictions and role axioms.
10. Connecting instances by assigning the relation between them. This can be cause-and-effect (as error factors and the malfunctions they lead to), clarifying the nature of the relation (mutually exclusive, compatible) and others.

During input, KBMS should check the input data for correctness and not move to the next stage until a required amount of information has been entered during the previous stage.

After the expert has entered all the information, the inference engine should be launched. This engine fills the knowledge base with additional assertions, logically following from the entered information – exactly for this the role and terminological axioms are described.

The last step is to check the formed knowledge base for integrity (consistency).

Machine elicitation of knowledge

The process of automatic elicitation of knowledge from large stores of information, for example, from the internet or from libraries, can be looked at separately.

Here, we need to separate automatic knowledge elicitation from unstructured sources and that from structured sources.

In the first case, the same features and difficulties that occur when working with a real-life expert during elicitation are made worse by the fact that, when working with unstructured sources, the "elicitor" program perceives all the words it comes across as just a set of symbols and has no idea about their semantic meaning. In connection with this, during a simple search using keywords, the resulting material cannot be guaranteed to be relevant (the searched word may not be the subject of the text but instead simply found there). Therefore, for a more reliable search of unstructured texts, natural language processing programs, for example, are used. These are capable of not only producing a parsing of sentences but can also build a semantic network based on it.

The other task is to search and extract materials on pre-structured data. Here, we are talking about knowledge existing in an open base network, on which a search can be gradually conducted. Examples of this are the portals DBpedia.org and Wikidata.org. The process of sending requests to such stores we will discuss later. For now, we will describe the work logic.

Here, the work begins not by listing the most important concepts, but by defining one main example.

A request is sent to a knowledge base to find the element of interest to us, for example, a robotic arm. If it does not exist, then the grammatical form of the searched element should be changed, a synonym selected, or a search conducted on another open knowledge base.

When the element is found, knowledge elicitation about it means searching on all relations in which it takes part:

- request for all classes to which it belongs;
- request for all literal properties and their values;
- request for all object properties and instances which it is connected to by these object properties;
- request for all synonyms and other names of the same searched element, and repeating with them all specified steps.

As a result, we get a number of assertions on the searched element in the form of assertions.

Knowledge formation

Knowledge can be obtained not only through conversation with an expert or by analyzing structured/unstructured sources, where it is already presented in some (at least implicit) form. Knowledge can be formed directly from data in which it was not originally embedded. This means, for example, data in the form of a table/graph of the dependency of productivity on the temperature and pressure in the circuit of the object.

In this case, it is about forming knowledge – the process of analyzing data and identifying non-obvious patterns using a special mathematical apparatus and software tools.

A good example of this process is training a neural network on some tabular data, that gives as a result models of the dependency between different system parameters. This can also include various methods of dependency approximations, cluster analysis, and others.

Methods of automatic hypothesis generation, for example, the JSM-method, can be placed in a separate group. It is based not on purely mathematical calculations like neural networks, but on the so-called formalization of plausible reasoning, at the center of which are logical operations.

In the framework of knowledge formation, we are interested most of all in two subproblems, each of which we are required to solve: the problem of inferring new (correct) knowledge and the problem of removing contradictory (incorrect) knowledge. In both cases, we will use rules for this on base of which the formation is made.

PART 2. STRUCTURIZATION. TYPES OF LOGIC. TRIPLETS

Concept of structurization

After elicitation, knowledge is stored in an implicit (**unstructured**) form, such as outlines, tables, interview notes, text files, etc. This may be enough to carry out human activity based on this knowledge, but it is not enough for computer systems. Why?

Since practical use of knowledge requires that it be well-ordered (structured), a human is forced to engage in **structuring** while learning and reflecting on the material – in other words, highlighting important connections in a data stream and forming assertions ready to be translated to knowledge representation languages.

Structuring usually results in diagrams and graphs reflecting the nature and number of connections between elements. Likewise, it can also give tables, formulas, and so on.

However, a human usually performs this operation implicitly (if not dealing with the building of diagrams). They are not taught how to order the contents of their thought process, but each person nevertheless does this in his head. Thus, unstructured material will be enough for work, since the person can elaborate it himself.

However, software agents by default do not have this ability, therefore the ability to structure or formalize needs to be partially or completely realized by a program, or already formalized material to enter into the knowledge base needs to be prepared. Next, we will look at both of these paths. In any case, a knowledge engineer experiences the need to solve the problem of transferring knowledge from an unstructured form to a structured one.

The presentation forms resulting from structuring can still not be used directly by the software agent, but the programmer working with the knowledge bases is able without expert help to translate knowledge to a knowledge representation language.

Let’s now discuss the process and ways of structuring by groups.

Structurization in graphic representation

The first and, possibly, the most well-known tool for structuring knowledge is mind maps. They are widely used for mapping conceptual structures in teaching, when designing software systems, in business and in a wide range of other fields.

These maps (when composed correctly, of course) are very illustrative. Their peculiarity is that the main object of study is located at the center of the map as the main concept and from it grow branches of the main areas of consideration, followed by a further nodal structure. An example is the map in Fig. 4.

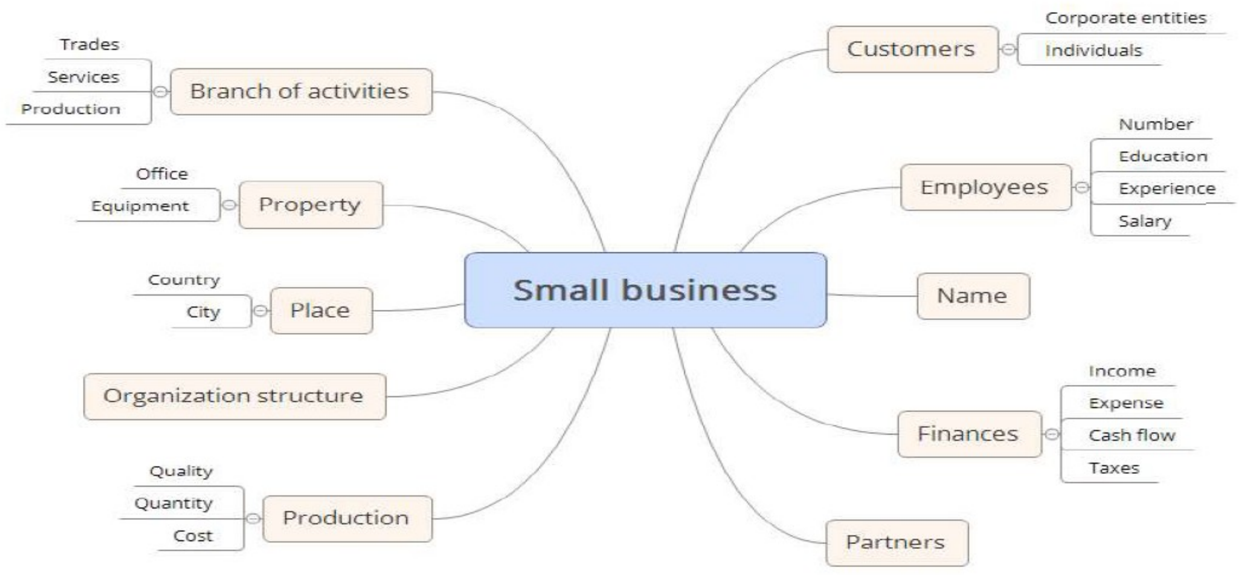


Fig. 4. Mind map.

At the first level of the generalization, we find the elements “Employees”, “Customers” and so on, while at the second, we find “Number”, “Experience” and so on. In total two levels are shown.

At the first glance these maps are simple, but developing them requires the skills of systematic analytical thinking. The main rules of constructing these types of maps are as follow.

1. Rule of uniformity (at each level there should be no concepts which are differ from others, not one should be “odd”).

2. Rule of gradual generalization: the transition to the next level of generalization should be understandable even for a reader who is not a specialist in the field.

Furthermore, there are also recommendations for building:

- Fonts of different sizes should be used when mapping concepts of different levels;
- colors are useful for highlighting branches and levels;
- visual images and pictures also increase the visual impact of a mind map.

The presented in Fig. 4 map contains some typical omissions and errors, for example:

- there are too many objects at the first level, which makes perception of this map difficult;
- these objects have a different level of generalization, i.e., the concepts are too dissimilar;
- the branches have a different depth of detail;
- images and colors are not used, but are important in mind maps.

Practice shows that students begin to understand the particularities of radiant (centered or hierarchical) thinking after the third or fourth constructed map with strict analysis of their mistakes.

However, the drawback of mind maps in terms of structuring knowledge is the fuzzy denotation of relations between the elements, since they are more adapted for demonstrating a tree structure of arbitrary fragments of knowledge.

In this regard, concept maps (c-maps) can provide greater unambiguity and visual effect. These concept maps are made up of nodes and directed named relations connecting these nodes. The connections can vary in type, for example, “is”, “has the property of”, “leads to” and so on. Any development of such a map involves analyzing the structural interactions between individual concepts of the subject area. A concept map is represented in the form of a graph, where vertices describe concepts (classes or their instances), and the directed named edges connecting these vertices are the relations (connections).

In the simplest case, building a concept map comes down to the following steps:

- determining the context by asking a specific focusing question that determines the main topic and borders of the concept map;
- selecting classes (concepts) – basic terms (concepts) of the given subject area (usually no more than 15–20 concepts);
- building connections between elements – determining the relations and interactions;
- streamlining the graph – clarifying, eliminating unnecessary connections, removing contradictions.

By building c-maps when creating knowledge bases or expert systems, specialists get the most complete idea about the subject area. It is worth emphasizing again that c-maps are not only the goal but also the method for a deeper understanding of the specifics of the subject area. While building c-maps the semantic connections of our memory interact with visual information, the connections are rebuilt, generating, in turn, new knowledge.

Fig. 5 presents a concept map for the education system of a city, developed within the System project of Moscow [Gavrilova, Onufriev, 2016].

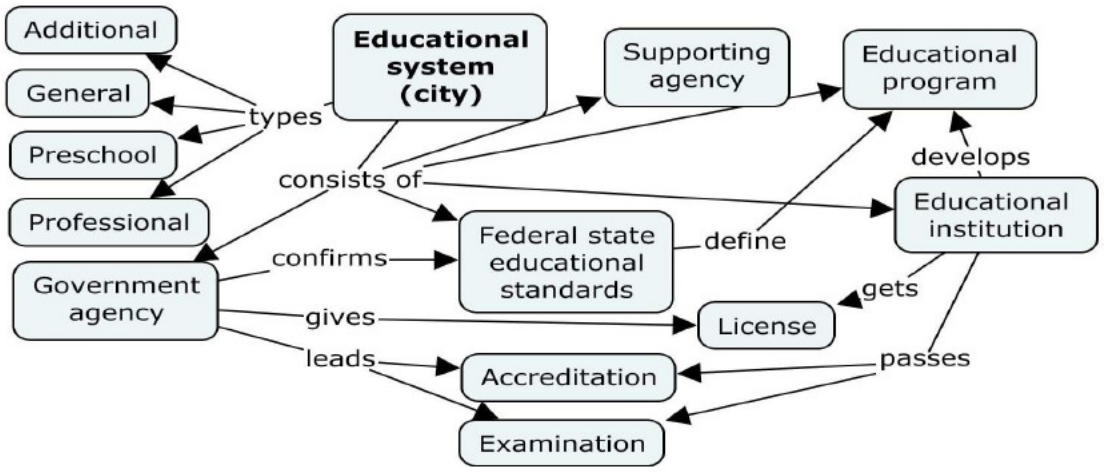


Fig. 5. Example of a concept map.

The most common mistake when creating c-maps is the incorrect marking of relations between elements: it may be that the generic term is improperly defined ("maple" – is a type of – "forest") or that it lacks useful information ("Electronic skin" – is a type of – "skin").

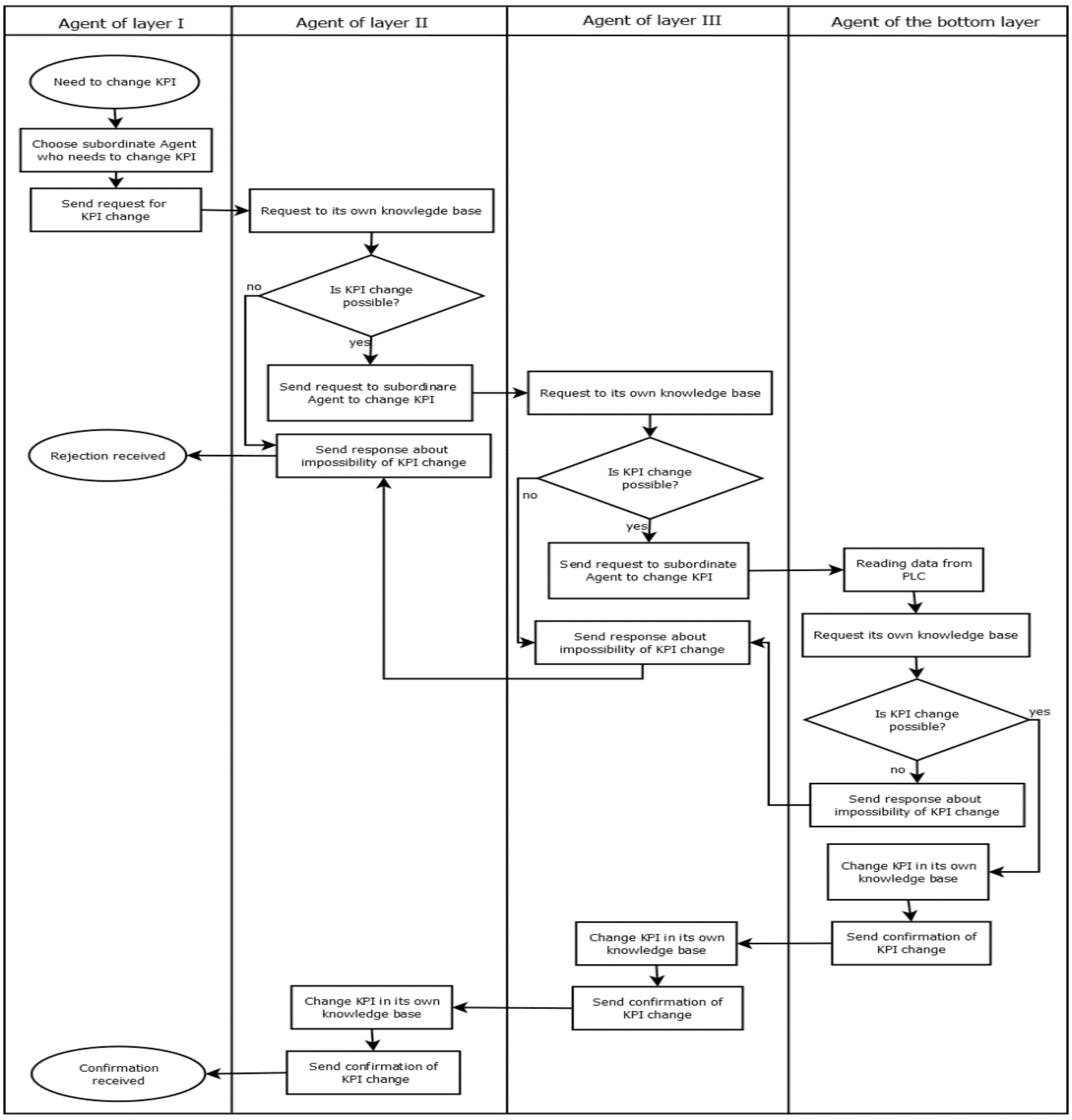
In order to construct a c-map on a random area of knowledge, its skeleton should first be constructed. It is built out of the fundamental classes – the most important concepts. After this, the most frequently mentioned individuals are designated, which are immediately associated with classes and with each other. Next comes the stage of designating the rules: role and terminological axioms.

This way can be seen the process of obtaining knowledge is in general.

Among graphic structuring tools can also be mentioned entity-relationship diagrams (ER-models), which are more focused on designing databases, but nevertheless, they offer visual depictions.

In order to structure cause-and-effect relations, a so-called argument map can also be used, which can be called a particular case of c-maps with relations of "because", "however", "but", and statements as nodes.

In the case of describing how- and when-knowledge, and in any situation where it is required to indicate the sequence of actions and conditions, tools such as a sequence diagram, swim lane diagram (Fig. 6), Gantt chart can be used.



Agent of layer I
Agent of layer II
Agent of layer III
Agent of the bottom layer
Need to change KPI
Choose subordinate Agent who needs to change KPI
Send request for KPI change
Request to its own knowlegde base
Is KPI change possible?
no
yes
Send request to subordinare Agent to change KPI
Send response about impossibility of KPI change
Rejection received
Request to its own knowledge base
Is KPI change possible?
no
yes
Send request to subordinate Agent to change KPI
Send response about impossibility of KPI change
Reading data from PLC
Request its own knowledge base
Is KPI change possible?
yes
no
Send response about impossibility of KPI change
Change KPI in its own knowledge base
Send confirmation of KPI change
Change KPI in its own knowledge base
Send confirmation of KPI change
Change KPI in its own knowledge base
Send confirmation of KPI change
Confirmation received

Fig 6. Sequence diagram.

Fig. 6 demonstrates the expressive ability of such diagrams to describe sequences and instructions: it presents an algorithmic model developed by the authors for the interactions of agents in a multi-agent system for production control [Kovalevsky, Onufriev, 2019].

A diagram of classes and other types of presentations used in UML-diagrams are worth a separate mentioning. Their description is easy to find in relevant sources. However, when using UML, it is important to constantly keep in mind the semantic value of all the elements used. For example, the relation of aggregation in the class diagram can be considered to be equivalent to the relation “used in” or “includes” in c-maps.

Text structurization

At the same time, the most flexible and, likewise, customary remains structuring knowledge as text. Here, we are not limited in tools. However, if we use this form, it is more logical to choose a text for this which would be most suitable for the subsequent formalization, in other words transferring to code in knowledge representation languages.

The basic form which we will use is the triplet form. In fact, this is an **assertion** which is clearly separated into three (thus the name) parts: subject, predicate, and object.

Here the subject is analogous to the subject known in the syntactic analysis of sentences. Important: the subject is not a sentence but an assertion. The subject is expressed by one word or by a word combination.

The predicate, which, as was already stated, can take the name “role”, “relation” in different sources, corresponds either to the verbal predicate in an assertion (“can cause”, “prevents”, “has”), or as the verbal part of a compound nominal predicate (for example, “is”, “was”, “is considered”).

The object, which is sometimes called the “value of a property” (since a property takes the role of predicate) corresponds either to a supplement (minor sentence) in the case of a simple verbal predicate (for example, “skidding”, “book”, “installation №2”, “car”), or as the nominal part of a predicate in the case of a compound nominal sentence (“operation”).

Triplets made up of these elements are given below:

Type of predicate	Subject	Predicate	Object
Verbal	braking	can cause	skidding
	damage to motor №3	leads to	necessity of landing
Nominal	removal	is	an operation

So, any assertion of any complex form can be presented by a finite number of triplets. However, in the case of preparing materials for formalization, the following points should also be taken into consideration.

1. Despite the rules of declination in different natural languages, all three of the elements in a triplet **should be in the nominative case**, since otherwise it will be necessary to introduce into the KBMS a module which, despite the different endings, will recognize the same substance in words with the same stem. By default, they will be perceived as being different.
2. As was already mentioned, any triplet contains exactly three elements. Tools for processing knowledge representation languages recognize them by the presence of a space between them. In connection with this, any triplet should contain **exactly two "space" signs**.

Taking into account these remarks, the following triplets were received from the table above:

breaking canCause skidding

damageToMotor3 leadsTo NecessaryLanding

removal is anOperation

Thus, we have a list of assertions in the form of triplets, which can then be formalized.

It is worth noting that c-maps can be divided into triplets very organically. Furthermore, an export of the map in the form of triplets can be provided in c-map editors (for example, "CMapTools" – an open source software).

When creating triplets, an analyst chooses themselves which properties to use and which individuals/classes will be needed for this. And while we are not restricted by anything it is however recommended to use maximally unified predicates: inOrderTo, causes/isCausedBy, consistsOf, and so on, to simplify future use of the triplets. In order to clarify and introduce details (how it is used, how it is caused) additional triplets are to be used, apart from unified ones. However, for standard relations, it is better to use those given above since, when working with a big knowledge base made up of thousands of triplets, the variety of relations makes it significantly harder to search with it and make queries.

Discussed below is the situation where more complex assertions need to be presented in the form of triplets, for example: "The lecture on knowledge engineering is at 2:00pm on Monday in lecture hall 310 and at 4:00pm on Thursday in lecture hall 314". When attempting to create such a series of triplets:

lecture – time → 2:00pm, lecture – time → 4:00pm

lecture – place → 310, lecture – place → 314,

we find ourselves in the situation where time "2:00pm" is not at all connected to place "310", due to which it becomes impossible to understand from these triplets at what time and in what place a lecture will take place since they are not grouped in any way.

The solution to this problem is to create some grouping elements, which are intentionally denoted in Fig. 7 as "????", since it matters little what will be contained in them. If this type of node has no name and is used exclusively for connecting, then it is called a **blank node**.

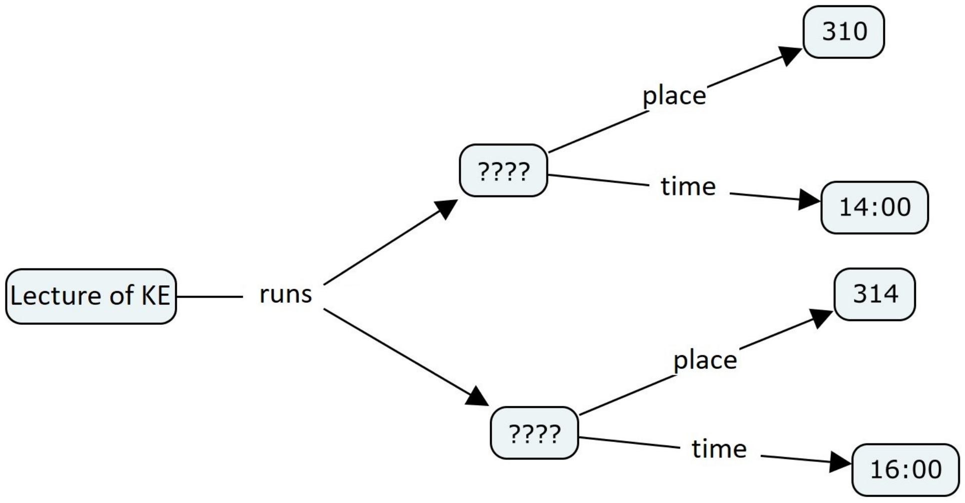


Fig. 7. Grouping nodes

The next important point has to do with the fact that any class can be considered to be the totality of its individuals or as the concept itself. However, in triplets, a class is always used as a concept. Since the individuals (class instances) or the classes themselves can act as the subject or object of a triplet, it is often tempting to say, for example, that the class Student is connected to the object "Knowledge engineering" with the help of the predicate "attendsCourse". This, however, would be a mistake because clearly what we are trying to say with it is not "the class Student attends a KE course", but "all instances of the class Student (i.e., all students – individuals of this class) attend a KE course". The difference is significant. If we want to talk about "all individuals of a class", we need to use rules (which is talked about below).

The class Student itself is used very limitedly and carefully in triplets, since using the class Student is equivalent to using the concept "student". Thus, if the subject of a triplet is the class Student, then its predicate can be either the connection subclass-class (subClassOf), or other relations with classes (for example "equivalent class"), or a defining relation (which will be examined in the section "Ontology"), or the features of a class as a set (for example, "class power").

In order to create more complex logical expressions (including rules), logical languages are needed.

Structurization using formal languages

Propositional logic

When structuring knowledge, the use of well-known formulas and mathematical notations should not be disregarded, since the task is to describe unstructured knowledge as clearly and systematically as possible. However, the alphabet of simple mathematics is not complex enough to describe the most frequently seen assertions with familiar formulas. It has other goals.

The formulation of assertions and their description is the work of logic, more specifically, its individual divisions, e.g., mathematical logic. As part of it, in turn, specific languages can be considered. Let's discuss three of them: language of propositional logic, language of first-order logic and language of description logic.

The simplest of those listed is propositional logic. Its language is characterized by the fact that its elementary unit is a statement: a coherent expression, for example, "Moscow is the capital of Russia". Propositional logic does not operate with individual concepts: it is more focused on analyzing the interconnection of statements.

This leaves a mark on its language, which contains the following elements:

- negation (¬): $\neg a$ is true when a is false and vice versa. Note that any statement must be either true or false;
- following or implication (→): for example, $a \to b$ means that from statement a follows (logically) statement b;
- disjunction ∨, logical OR. For example, $a \lor b$ is true if the first or the second statement is true;
- conjunction ∧, logical AND. For example, $a \land b$ is true if both the first and the second statement are true.

The following complex statement from *The Little Prince*, Saint-Exupéry, can be taken as a construction for notes: "If I order the general to turn into a seagull and he fails to follow the order, then the fault is mine not the general's" [Zyuz'kov, 2015].

In order to write it down, it is necessary to make four statements: a – "I order the general to turn into a seagull", b – "the general can turn into a seagull", c – "I am at fault", d – "the general is at fault". As a result we get:

$$a \land \neg b \to c \land \neg d.$$

Note that each statement written in this language represents not a class, not an instance, but a whole triplet at once. Therefore, it is impossible to describe the connections between the examples themselves and with the classes in this language.

Another disadvantage of the language of propositional logic, which is significant in terms of structuring knowledge, is the lack of tools for working with generality: a statement concerning some set of elements does not differ in form from a statement of one element.

However, this language can fully be used as an intermediary on the way to adopting more complex tools for describing facts and axioms. For example, thanks to it we can formulate the rule:

$$[\ (\mathrm{a} \to \mathrm{b}) \land \mathrm{b}\ ] \to \mathrm{a},$$

which can then be checked for creating false assertions, and using certain tools, the fallibility of this rule can be made certain.

Languages for describing sets and relations

The next important step to correctly understanding logical processes in the area of knowledge is to become familiar with the set theory, especially in terms of formal descriptions of the sets themselves and the relations between their individual elements, since later on it will all be used when developing projects.

Any class should be understood either as a concept itself, or as a set of individuals included in the class (at the same time, we remember that when writing in triplet form, the first is always meant, and when writing with sets – the second). In connection to this, if a class A is a subclass of class B, then in terms of sets, we can say that $A \subseteq B$ or set A is included in (is a subset of) set B, in other words: all individuals of class A are also individuals of class B. This inclusion is not strict, and any set is a subset of itself.

A strict inclusion is denoted with the symbol $\subset$, and if $A \subset B$, then A is its own subset of B, i.e., elements exist which belong to the latter and not the former. In terms of classes, this means the existence of individuals of the second class not belonging to the first.

The operations that interest us here the most are union $\cup$ and intersection $\cap$, as well as the negation $\neg$. For example, $\neg A$ in terms of classes means the totality of all individuals not belonging to class A, and if we say that $\mathrm{A} \cup B \equiv C$, this means that all individuals belonging to either class A or class B are also individuals of class C.

Here, we are especially interested in De Morgan’s laws, for example:

$$\neg(A \cup B) = \neg A \cap \neg B,$$

because then $\neg C = \neg A \cap \neg B$, i.e.: all individuals not belonging to class C cannot belong to class A or class B. It would seem we are talking about obvious conclusions, but they have mathematical accuracy and are easily formalized into programing code, which can check hundreds of knowledge base classes for the presence of any contradictions.

It is important to also look at the relations that appear in set theory.

We are mainly interested in binary connections between objects of sets, since this will subsequently be applied when formalizing connections between individuals in knowledge bases.

The relation between two individuals is described as: $\rho = <x, y>$. Next, the content of ρ should be clarified, for example, $\rho = \{<x, y> \mid x$ is the father of $y\}$. This has a clear connection to the language of triplets.

In order to analyze and modernize knowledge bases, **inverse** relations are required. Thus, we can determine that ρ^{-1} $\{<x, y> \mid x$ is the son of $y\}$. In other words, if the knowledge base contains some x and y which are connected by one of these relations, then it is possible to conclude that y and x are inversely related.

The compositions of the relations are also of interest. Thus, if we introduce the relation z $\{<x, y> \mid x$ is the brother of $y\}$, then the composition $\rho \circ z <x, y>$ will denote "x is the uncle of y".

At the same time, there is a theorem on inverse compositions:

$$(\rho \circ z)^{-1} = z^{-1} \circ \rho^{-1}$$

This is well-illustrated by the concept map in Fig. 8, where the names of the inverse properties are highlighted in red.

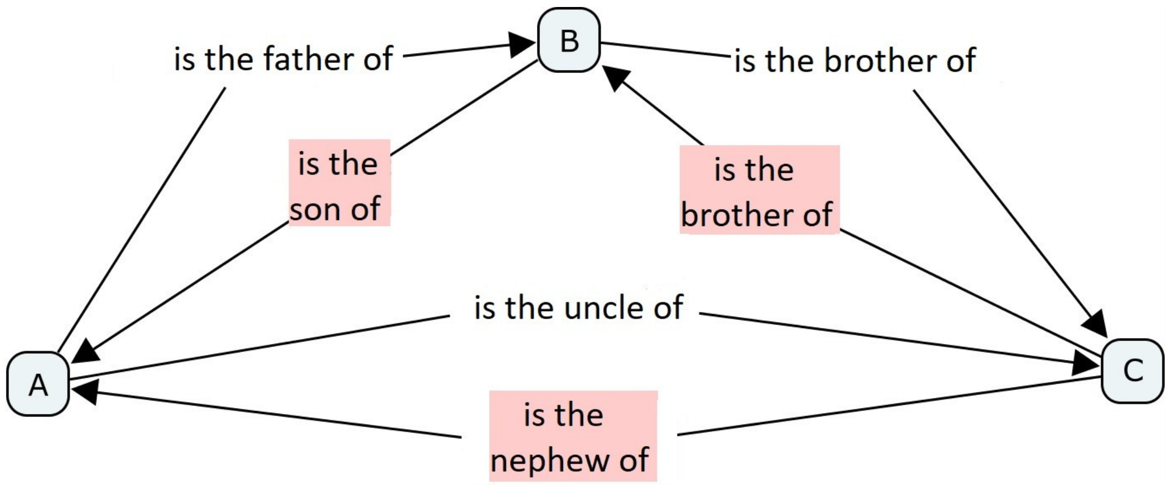


Fig. 8. Compositions of relations and inverse compositions.

Besides the composition of properties and inverse properties, attention should also be given to **subproperties**. The main idea can be explained in the following way:

$$\rho <a, b> \& \rho \subseteq z \quad \rightarrow \quad z <a, b>.$$

In order to solve the following problems of creating knowledge, it is important to also pay attention to the existing types of relations, since this gives additional material for a logical conclusion and for checking for contradictions.

The first type is **transitive** relations, an example of which is a relation like $\rho <x, y>$, from which it can be concluded from $\rho <a, b>$ and $\rho <b, c>$ that $\rho <a, c>$. Otherwise, this can be written as $\rho = \rho \circ \rho$. A classic example of a transitive relation is $\rho = \{<x, y> \mid x \text{ is a subclass of } y\}$.

Likewise, other relations of interest to us are the so-called **symmetric** relations. This means that from the connection $\rho <a, b>$, it logically follows $\rho <b, a>$, in other words, the relation is inverse to itself: $\rho = \rho^{-1}$. An example of this kind of relation can also include "is related".

Here we can also speak of an **antisymmetric** relation, where from the connection $\rho <a, b>$, it logically follows the impossibility of the connection $\rho <b, a>$. Introducing this rule into the knowledge base can be used for searching for contradictions within it.

A **reflexive** relation implies that for any element we get $\rho <a, a>$.

A **functional** relation implies the uniqueness of the object to the subject. Thus, if an individual is connected by a functional relation to two elements, then from this follows the identity of the latter:

$$\rho <a, b> \& \rho <a, c> \rightarrow b = c.$$

First-order logic

The language of predicate logic, also known as the language of first-order logic (FOL), is devoid of the disadvantages characteristic for the propositional logic language.

Its first feature, in fact, is the use of predicates. The term "predicate" from first-order logic should not be confused with the term "predicate" from the set "subject", "predicate", "object" in triplets.

The predicate will be a function with an arbitrary number of input parameters and which gives as output either "True" or "False". Thus, if the predicate *E*(*x*) means "*x* is equipment", then the value of the predicate *E*(engine) is "True".

The predicate in the first-order logic always plays the role of not just a part of the sentence predicate but of the whole sentence predicate. For example, if *x* is any individual, and *S*(x) means: "*x* is a student", then the construction *S*(Nikolay) means that "Nikolay **is a student**". If we look at the triplet "Nikolay is a student" as a triplet, then its predicate (in terms of triplets) is only one word, "is".

Pay attention to the fact that the semantic content of the predicate *S*(*x*) is selected randomly according to the set task and the subject area.

In this example, the **unary** predicate *S*(Nikolay) corresponds to the whole triplet, correlating the individual "Nikolay" with the class (set) "Student". Overall, any unary predicate can be reduced to this: to assigning an individual to a set (i.e., to the class). This even has to do with verbal predicates: to say that instance *x* runs, it should be assigned to the predicate *R*(*x*) – beings which can run (or are running in that moment). Note that this is nevertheless assignment to a class.

Binary predicates like *P*(*x*, *y*) can be used in order to describe connections between elements, in our case, individuals. For example, *L*(*x*, *y*) can mean "*x* launches *y*" – i.e., in this case, the binary predicate <u>denotes a whole triplet</u>. This also agrees with the relations discussed earlier in set theory.

The number of parameters and the semantic content of the predicates is determined by the user themselves, therefore any predicate should be previously described verbally.

The second feature of the language of first-order logic is the presence of certain denotations called quantifiers:

–∀ – universal quantifier;

–∃ – existential quantifier.

Thanks to these quantifiers, we can talk about not only specific individuals, but also about a set of individuals, selected according to some criterion. For example,

$\forall x.R(x)$ means: "anyone who can run",

$\forall x.L(x, \text{engine})$ means: "everything that launches engine 5",

$\forall x.[\ R(x)\ \&\ L(x, \text{engine})\ ]$ means: "everything that runs and launches engine 5".

Next, we can already construct statements that will be applicable to the entire specified set.

Note that in the shown examples, each predicate describes a criterion according to which a set of elements satisfying it is chosen. Meanwhile, a combination of several such predicates is possible using logical connectives, for example, the logical AND (symbol "&").

Universal quantifiers are used to derive new knowledge based on the old, as well as for checking existing knowledge for contradictions.

Existential quantifiers are used in our tasks not that often and can be used for the second problem relating to checking for inconsistent assertions.

The rules relating to predicates should also be mentioned separately.

The first ones describe a method for transferring from one quantifier to the other:

$$\forall x.A(x) \equiv \neg\exists x.\neg A(x);\ \forall x.\ \neg A(x) \equiv \neg\exists x.A(x),$$

$$\exists x.\ A(x) \equiv \neg\forall x.\ \neg A(x);\ \exists x.\ \neg A(x) \equiv \neg\forall x.A(x).$$

The second set of rules describes expanding logical addition and multiplication when acting with predicates:

$$\forall x.[\ A(x) \lor \mathrm{B}(x)\ ] \equiv \exists x.A(x) \lor \exists x.B(x),$$

$$\forall x.[\ A(x)\ \&\ B(x)\ ] \equiv \forall x.A(x)\ \&\ \forall x.B(x).$$

Language of description logic

A special role can be assigned to the following tool: a set of languages of description logic – rather complex, and therefore perfectly adapted to the tasks of developing and describing knowledge bases.

The basic language of description logic is ALC (Attributive Language with Complement). It is aimed at describing the relations between classes and describing the connection of instances and classes, therefore, it initially contains not only the concepts of class (e.g., *C*), individual (e.g., *a*) and relation (e.g., *R*), but also the constructions of type: *a*:*C* (individual belonging to a class), *aRb* (connection of two individuals by a relation), the earlier described elements of the system of facts ABox, as well as $C \equiv D$ (equivalence of classes) and $C \subseteq D$ (inclusion of one class in another), elements of the terminology TBox. The same is true for the role axioms Rbox.

The classes $\perp$ and T are also introduced, where the first denotes an empty class (empty set), and the second – a class-universum, whose subclass will be any of the possible classes.

Furthermore, within the language, there are so-called **universal restrictions** like $\forall R.C$ and **existential restrictions** like $\exists R.C$. In both cases, *R* is the property and *C* is the class. Here, "restriction" is understood as the condition itself which divides individuals into those that satisfy it and those that do not, as well as the set of satisfying individuals itself. In the latter case, the concept of restrictions is close to the concept of class.

In order to deal with universal and existential restrictions, it is necessary to introduce the concept "successor by role (by property)". If we have the triplet *aRb* (i.e., *R*<*a*,*b*>), then the individual *b* is called a **successor of individual *a* by property (role) *R*** (or ***R*-successor**).

Now, $\forall R.C$ can be defined as a set of individuals where all *R*-successors belong to the class *C*. In other words: those without one *R*-successor not belonging to the class *C*. An example of this can be the restriction ∀hasAuthor.Poet – is a set of all individuals whose authors are only poets (there is no author who does not belong to the class Poet). In the language of predicate logic, this can be written as: ∀*x,y* [hasAuthor(*x, y*) & Poet(*y*)].

$\exists R.C$ can be defined as a set of individuals among *R*-successors of which there is at least one individual from class *C*. For example, ∃writtenWorks.PhDThesis – is a set of all individuals whose written works contain at least one belonging to the set (class) of PhD theses. In the language of predicate logic, this can be written as: $\forall x \exists y$ [P(x, y) & PhDThesis(y)].

There is yet another simpler construction: *R.ind*, where *ind* is the individual and not the class, as in the two given examples. This type of restriction describes a set of individuals whose *R*-successor is a specific individual. For example, hasAuthor.PushkinAS is a set of all individuals whose author is A.S. Pushkin.

When adding a new functionality to the ACL language, its name also changes in order to reflect its expressive ability. For example, when adding to the standard three restrictions numerical ones like $\geq nR$ and $\leq nR$, which denote a set of all individuals having no less than (no more than) n *R*-successors (this is called a **cardinality restriction**), and the letter *N* is added to the name. When adding the restrictions $\geq nR.C$ and $\leq nR.C$ (a set of all individuals with no less than (no more than) n *R*-successors from class *C*, the letter *Q* is added (Qualified cardinality restrictions). This can result in a very complex and flexible language. We recommend becoming familiar with the extensions of ALC logic.

An important feature of such descriptions is the fact that this method can also be used to describe knowledge which does not yet exist in the data base but which will be derived from existing knowledge or added by an analyst in the future. Such descriptions work on the possibility of carrying out knowledge base training.

Describing rules

So, a rule is any assertion (and, therefore, triplet) which deals not with specific individuals but with their set. The latter, in turn, can be described as the association of individuals to one general class or by the fact that they all satisfy a general requirement, called restriction.

In connection to this, all rules can be divided into two types: those applicable to a class and those applicable to a set of individuals designated by a restriction.

The simplest form of the rules from the first group is "**class-class**". This name should be understood as: "If an individual belongs to a specific class, the same individual at the same time belongs to another class".

However, this exactly corresponds to the occurrence of one class in another, therefore these types of rules are the simplest and are expressed as a triplet

$$\text{Class1} — \text{isSubclassFor} \rightarrow \text{Class2}$$

and can be interpreted as "any individual belonging to class 1 also belongs to class 2".

These rules can also be applied when there is a formulation with a verb, for example, the rule "any robotic arm requires precise settings" can be formulated as a triplet

$$\text{roboticArm} — \text{subclassFor} \rightarrow \text{EquipmentRequiringSettings}.$$

When using methods of predicate logic, this assertion can be written, for example, as (denotations for predicates are chosen randomly)

$$\forall x.[\ \mathrm{M}(x) \rightarrow \mathrm{Sett}(x)\ ].$$

One interesting fact to note is that any triplet whose subject is a class is often a rule.

Note also the rules for overlapping and joining classes such as

$$\forall \mathrm{x}.[\ \mathrm{A(x)}\ \&\ \mathrm{B(x)} \rightarrow \mathrm{C(x)}\ ];$$

$$\forall \mathrm{x}.[\ \mathrm{A(x)} \vee \mathrm{B(x)} \rightarrow \mathrm{C(x)}\ ].$$

They are useful in that they create a background for applying De Morgan's laws mentioned earlier to produce new rules.

The rules that use existential quantification make it possible to find triplets in the knowledge base which are contrary to the rules and to exclude them. For example, if we have the rule

$$\forall x\ \neg\exists y[\ \mathrm{C}(x)\ \&\ \mathrm{B}(x, y)\ ],$$

then finding the connection B(x,y) in the knowledge base is an error, and the triplet describing it should be deleted.

Rules with the following form work in the same way:

$$\neg\exists x[\mathrm{C}(x)\ \&\ \mathrm{B}(x, \text{soundSensor})].$$

Another example is the **class-property** rules. They also consider a set of individuals restricted to one class, and prescribe them certain properties/relations. For example, we suppose that any synaptic nucleus matrix (in the considered area) has 3 dimensions.

The indicated class will be given the name NucleusMatrix. Note that it is **unacceptable** to be comprised of a triplet

$$\text{NucleusMatrix} — \text{hasDimensions} \rightarrow 3,$$

since this property refers to all individuals of the class and not to the class itself as a concept, which was discussed above.

In order to express that any individual of the class NucleusMatrix has a dimension of 3, we use the language of first-order logic. For this, we introduce the predicate NM(x), which signifies that individual x belongs to the class NucleusMatrix. With this designation, the construction $\forall x.\text{NM}(x)$ describes a set of all individuals belonging to NucleusMatrix. Now we can replace the incorrectly written triplet (shown above) with the correct one:

$$\forall x.\text{NM}(x) – \text{hasDimensions} \rightarrow 3.$$

To represent it fully in the language of first-order logic, we will introduce the binary predicate Dim(x, y), meaning "individual x has y dimensions". Then we get $\forall x.[\ \text{NM}(x) \rightarrow \text{Dim}(x, 3)\ ]$.

The rules are displayed on a concept map in the following way (Fig. 9).

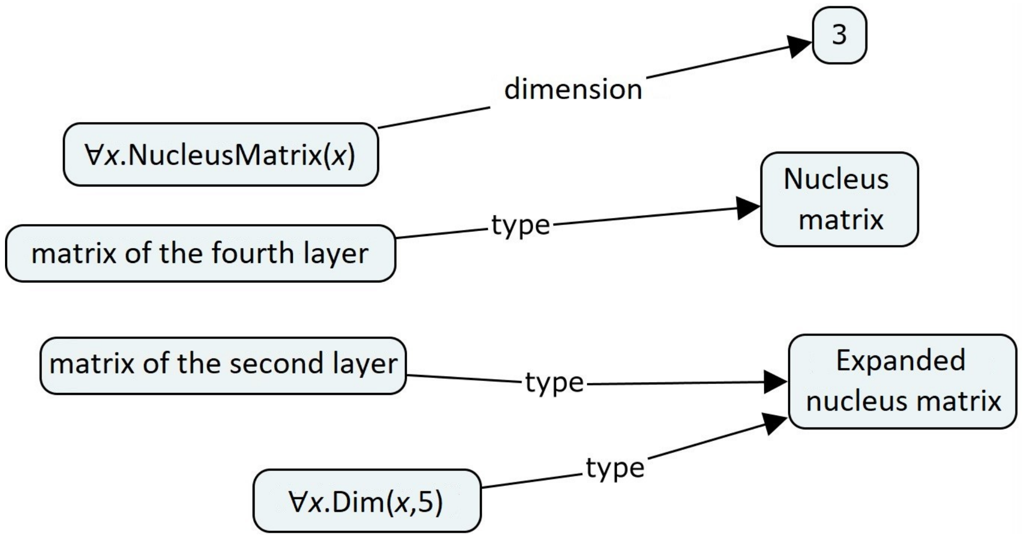


Fig. 9. Introducing rules to a concept map.

Rules of the second type, in which the set subject is determined not by belonging to the general class but by an arbitrary restriction, are more complex since the analyst is free to select the subject constraint in the formulation of the rule.

The simplest of these are formulated in a way which is already familiar (from description logic): *R.ind*, which means "a set of all individuals related by the property *R* to the individual *ind*". If this is translated from the language of description logic to the language of first-order logic, we get

$$\forall x.R(x, ind).$$

After writing the subject in this way, it can either be assosiated with a certain class, or it can be assigned some value of a property.

In the first case, the rules obtained belong to the **property-class** type (unlike the class-property type rules described above). An example of this kind of rule is "anyone whose list of studied subjects includes research work is a fourth-year student":

$$\text{discipline.RW} \xrightarrow{\text{is}} \text{4yearStudent.}$$

An example of the second type of rules (**property-property**) is the following:

$$\text{discipline.RW} \xrightarrow{\text{discipline}} \text{undergraduatePractice,}$$

this means: everyone who has the discipline "RW" also has undergraduate practice.

Note that the subject of such assertion rules can be not only simple restrictions, but also the universal restrictions looked at earlier and existential restrictions, presented in the language of description logic. For example, "Everything that can lead only to malfunctions should be assigned to a class "Negative factor":

$$\forall\text{causes.Malfunctions} \xrightarrow{\text{is}} \text{NegativeFactor.}$$

We will also consider rules which concern uniqueness (which will also not be a finite list):

–uniqueness of an individual in the class:

$$\exists x A(x) \,\&\, \forall x, y(A(x) \,\&\, A(y) \rightarrow x = y);$$

–uniqueness of a follower by property:

$$\forall c,x,y(A(c,x) \,\&\, A(c,y) \rightarrow x = y);$$

– uniqueness like “Only individuals of class A are individuals of class B” (e.g., only fish belong to creatures able to breath under water):

$$\neg\exists x.\ [\ \neg R(x)\ \&\ B(x)\ ],\ \text{from which we get}\ \forall x\ [\ B(x) \rightarrow R(x)\ ].$$

The rules for describing relations are described in exactly the same way as it was described in the section on describing relations in set theory.

In general, any rules can be described using the languages of first-order logic and description logic thanks to the flexibility of the latter.

Ontology

Earlier the various methods for structuring knowledge were looked at. However, they do not tell about creating a comprehensive system, since they are aimed at arranging individual segments of knowledge. Now we need to look at the result of structuring as a single whole.

After structuring knowledge, we get a set of the following elements:

1. individuals (examples) – entities selected from a set of others in this subject area which are of the most value to the tasks which the knowledge base was created for;
2. classes – these are the concepts or sets to which the individuals belong;
3. relations (these are connections, roles, properties) – these are subdivided into **object**, denoting relations of class-class, individual-individual, individual-class, and **literal (datatype)**, denoting relations in which the object is a numeric or string type of value, for example maximum pressure (105), model (“Grunfos 300”), inscription (“Class for describing governing bodies”). Here we should point out a unique type of relation: this is the **determining** relations or relations of determination, thanks to which it is possible to not only designate for any class its generic term (parent class), but also to determine its distinguishing properties and their values through the generic term.
4. rules (axioms) – assertions directly about a set of elements. They are used to infere new knowledge or to check the whole consistent ontology for

contradictions. This can include role axioms and terminological axioms, as well as the other rules looked at in the relevant section.

Ontology – this is a computer-adapted form of describing a certain subject area which includes the set of classes *C*, their relations to each other *R* (including determining *I*) and the rules *A* (axioms):

$$\{C, I, R, A\}.$$

Note that individuals are not included as part of the ontology. Thus, the ontology is the backbone for filling the knowledge base with individuals in the future.

Ontologies can be classified according to various features. First, we will discuss their classification according to the content of the specified components.

The simplest type of ontologies from the ones we are interested in are **dictionaries**. They contain only two components: classes and their definitions (defining properties), which can here act as a literal property or as a semantic structure with object properties and their values. In the first case, the class "Bird" can be defined through the literal property "definition" and the value of the latter "a warm-blooded chordate covered with feathers". In the second case, the class "Birds" is defined as follows:

bird – defining property → type of covering – defining value → feathers.

It is clear that there can be several defining properties and their values.

Note that in dictionary ontologies, there are no other relations between classes besides defining ones, and axioms are also missing.

A more complex structure in terms of connections is **taxonomy**, which is a set of classes connected either by only a "subclass-class" relation, or also by defining relations. This is a hierarchical structure presented in the form of a tree. A clear example of taxonomy is the presentation of goods in online stores, where items are divided into groups (sets, classes) and then subgroups and so on:

Ultrasonic sensors — subclass → Sensors — subclass → Measuring equipment

A similar structure is **partonomy** – a set of instances (specific representatives of classes) connected to each other by only part-whole relations. Here it is very important to not confuse the relation “part-whole” with “instance-class”. The leg of a chair is part of the chair, but it is not a type of chair (as opposed to a fold-up chair). An analytical engineer represents a (type of) person but is not a part of a person (as opposed to a hand). For a more comprehensive understanding, it is recommended to study section 10.3 of the textbook by Albert Nikolaevich Knigin [Knigin, 2002].

The next type of ontology is a **thesaurus –** a dictionary which contains, along with literal and object definitions of classes, the terminological axioms discussed earlier, i.e., subclass-class relations and class equivalence relations. Relations of class synonymy can also appear here. There may also be indications of classes antonyms, homonyms, etc.

Next, the more rules and restrictions are introduced, the more *heavyweight* an ontology is called. In the most heavyweight, we can also find expressions in the language of description logic and in the language of first-order logic, and restrictions freely constructed by analysts. Of course, all axioms of the knowledge base are processed by a special software module, **Reasoner**, which has access to the rules and facts.

Furthermore, ontologies can also be classified according to the scope of their content.

Thus, the simplest type of ontology is **Task** ontology, in which the solution to a specific problem is described. In general, this includes classes connected to goals, tools of achievement (including equipment), methods of measuring results, key quality indicators – specifically for one selected task.

A slightly broader ontology in terms of content is **Domain** ontology. They describe general concepts (classes) of a domain, but without being bound to tasks. For example, metallurgy may be described as a whole, the occurring processes, methods for managing them, and type of personnel, etc.

At the junction of task ontology and domain ontology there is **Application** ontology. It looks at several tasks of one domain.

More general is the **Top-level** ontology (Upper ontology, Foundation Ontology), which can describe the most general concepts like Events, Processes, Tasks, and even Space/Time. Therefore, it can be used as a general template when developing more particular ontologies.

DBpedia as a tool for the machine structurization of knowledge

The use of ontologies can be considered on the example of DBpedia – a large international project, developed and sponsored by a community of users. The objective of the project is to structure information contained in another, also open, project, specifically: Wikipedia. The latter contains a large amount of useful information. However, due to its lack of structure (clearly, in terms of knowledge engineering), its use by software agents is made difficult.

The idea is to structure part of the information presented in Wikipedia in the form of triplets, then to describe it in knowledge representation languages. However, taking into account the variety of pages, this required creating a universal tool for all of them in order for the various pages to be structured in the same way. This structurization tool required a mechanism for grouping Wikipedia pages and it was found in the so-called Templates.

On the webpage https://en.wikipedia.org/wiki/Alexander_Pushkin, for example, if one switches to source code view, the following construction can be seen:

```
{{Infobox writer
| birth_name = Aleksandr Sergeyevich Pushkin
| image = Orest Kiprensky - Портрет поэта А.С.Пушкина - Google Art Project.jpg
| ... = …
| birth_date = 6.6.1799 (26.5)
| death_date  = 10.2.1837 (29.1)
| ... = ...
| ... = ...
}}
```

This is the so-called page template, whose name, in this case, is Writer. More detailed information about this is given on the page https://en.wikipedia.org/wiki/Template:Infobox_writer.

Wikipedia also implements a mechanism for obtaining a list of all pages that use any of the available templates. For example, on the resource https://en.wikipedia.org/wiki/Special:WhatLinksHere/Template:Infobox_writer?limit=500&namespace=0, you can see pages of everyone who was classified as a writer. Thus, the presence of a mechanism for separating into classes and individuals is easily noticed.

Furthermore, each template has fields. For “Writer”, as shown above, these are “birth_name”, “birth_date”, “image”, etc. Every page relying on a template has a table in the upper right corner. This table has two columns, thus it forms one triplet for each of its lines. Here the subject is the page itself (more specifically, the resource presented on it), the predicate is the content of the first column cell, and the object is the content of the second column cell.

Consequently, part of the Wikipedia materials are already structured, but this is still not a result of formalization. Therefore, it was then required to use information about Wikipedia templates and their fields in a way so that algorithms for the automatic formalization of knowledge could be developed (translating them to knowledge representation languages).

It is this mechanism (as with some others) which is implemented in the DBpedia project. It is described on the resource http://mappings.dbpedia.org/. The idea is to map the Wikipedia ontology, which includes the page templates themselves as the classes and the page template fields as the relation between classes, with the ontology of DBpedia, which also includes classes and relations (the datatypes included in it will not be discussed here).

The process of mapping the Wikipedia template with the DBpedia ontology means indicating,

1. which ontology class of DBpedia a specific Wikipedia template corresponds to;
2. which of the ontology relations/properties of DBpedia each of the template’s fields correspond to.

An important point here is the fact that Wikipedia is multilingual, which creates a potential (and often occurring) possible situation where templates are duplicated in different languages (like “Писатель” and “Infobox writer”). Since they, in fact, mean the same thing in content, both should be assigned to the same class on DBpedia. The same can be said about the multilingual duplication of properties (for example, “имя” and “name”). They should also be assigned to the same Ontology Property on DBpedia.

To implement the said above the following resource was used: http://mappings.dbpedia.org/. The mappings themselves (with the example of Mapping ru:Писатель – “writer” in Russian language) take the following form:

```
{{ TemplateMapping
| mapToClass = Writer
| mappings =
      {{ PropertyMapping | templateProperty = Имя | ontologyProperty = foaf:name }}
      {{ PropertyMapping | templateProperty = Оригинал имени | ontologyProperty = foaf:name }}
      {{ PropertyMapping | templateProperty = Псевдонимы | ontologyProperty = pseudonym }}
      {{ PropertyMapping | templateProperty = Имя при рождении | ontologyProperty =
      birthName }}
      {{ PropertyMapping | templateProperty = Дата рождения | ontologyProperty = birthDate }}
      {{ PropertyMapping | templateProperty = Место рождения | ontologyProperty = birthPlace }}
      {{ PropertyMapping | templateProperty = Премии | ontologyProperty = award }}
      {{ PropertyMapping | templateProperty = Сайт | ontologyProperty = foaf:homepage }}
}}
```

Example taken from the section Mapping ru:Писатель.

Note that “templateProperty” indicates the Wikipedia template field, “ontologyProperty” – the property from the DBpedia ontology, and “mapToClass” – the class of the DBpedia ontology.

After adding mapping, a script is launched after some time which finds all pages in Wikipedia made according to the specified template, and it formalizes the values of the fields into code in the knowledge representation language according to the DBpedia ontology.

At the end of the work, the systems obtain pages which are presented to users as such: http://dbpedia.org/page/Alexander_Pushkin. At the same time, in the source HTML code of such pages, links to the knowledge base code are found, for example: http://dbpedia.org/data/Alexander_Pushkin.n3.

Next, we will discuss the particular knowledge representation languages and the approaches and processes of translating knowledge into them.

PART 3. FORMALIZATION

Methods for formalizing knowledge

Knowledge in a format convenient for software agents (we are only interested in these types of formats!) can be presented in various ways. The process of translating knowledge into a machine-readable (and machine-applicable) form from any other form is the process of formalization (representation).

To complete this action, first of all, there needs to be non-formalized knowledge, and, second, to know which form this knowledge is going to be translated to.

There are certainly more than one of these forms.

The first of these is neural networks. As mentioned before, knowledge is stored in them in the mathematical form of representation. If we speak about neural networks without feedback, we mean weight matrices, each of which describes a topology of connecting neurons in the appropriate layer, and activation function matrices. Different rules and different assertions can be stored here but only implicitly. This, however, does not contradict the solution to formalization.

Neural networks are used in problems of processing mathematical (including those converted from graphic form) data, when the incoming data needs to be classified, or an evaluation, diagnostics, prognosis needs to be completed using this data, or to immediately calculate them and convert them to other signals.

Neural networks are a specific type of mathematical model. In a more general case, knowledge can be stored **as mathematical models**, which include all types: physical, chemical, geometric, and other forms. It can be presented as matrices and array data structures, as functions in programming languages, as project files of various modeling environments (Computer aided design – CAD), etc.

The following format of representation can be presented in languages for **describing production rules**, for example, Prolog. The latter, at the same time, offers a single environment for storing knowledge and for inputting, extracting, changing and using it for completing logical operations. Here you can set triplets with a simple declaration, for example, the triplet

Robot(arm)

means that the arm is a robot belonging to this class. This note more often than not corresponds to first-order logic.

We can also give the rule

Equipment(x) :- Robot(x),

which denotes the rule we are familiar with “Class-class”.

The follow-up request

?- Equipment(arm)

will have the meaning “TRUE”.

Another form of non-mathematical knowledge representation is **semantic web technologies** (hereafter SWT), which provides the maximum possible flexibility for the content of knowledge bases and can also be integrated into projects using other formats. At the same time, SWT are a very specific type of technology, developed by the WWW Consortium. Therefore, the most complete information with updates can be found on the official web portal.

To develop knowledge-based systems, it is required, of course, to combine various forms of knowledge representation and storage.

Next, we will discuss in more detail semantic web technologies, which are fully capable of claiming the role as the “heart” of knowledge-based systems.

Resources in semantic web technologies

Semantic web technologies are inextricably linked to the concept of Linked Data, which desires to bring order to the internet in the following way: right now it is a set of documents (including dynamically generated ones) linked to each other at best by hyperlinks, but not semantically related.

A semantic connection implies that, for example, from the page with the description of control system of some type, the programming agent (not only a human) can go to the page with the description of the components of this system, to the page where it is applied, with a description of the developer, etc. This is all possible when explicitly setting the semantic meaning of the connections themselves, when the connections are described and comprehended (i.e., not simply stated that the developer is shown in the link, but also described that there is a "developer", which characterizes this property, etc.).

The concept of Linked Data means exactly this type of connectedness, which implies that different users create web documents and apps, at the same time establishing a semantic connection with other already existing documents.

It is because of this that formalization using Semantic Web Technologies starts with the concept **resource**, which means an atomic unit of knowledge (class, individual, predicate), having its own description and being the subject of one or several assertions. In other words, everything worth examining in a knowledge base.

In connection to this, it is worth immediately indicating the elements which are not resources, since resources and non-resources are formalized in different ways. Numbers and constant-strings are not resources. If a knowledge base indicates, for example, the maximum range of some measurer, it will simply be a number (15.65). There is not need to explain anything about this meaning. It is not a separate subject to discuss. It is equal to a text comment for users on the same measurer: "a laser rangefinder is used for taking measurements". On its own, this string, enclosed in quotes, is not part of any other assertions and is only the object of the assertion (not the subject).

Any numbers and text strings within SWT are called **literals**. The latter, unlike any resources, do not have any **identifiers**, which serve as a unique name for resources.

Taking into account the idea of Linked Data, it is easy to understand the very first problem which users run into: this is the millions of resources which are publicly available and, therefore, should differ from each other in some way.

What is used to identify resources on the internet today? The unique identifier is URL – Uniform Resource Locator, more commonly known as "page address". However, this is not the only method which makes it possible to separate from one another the potentially endless number of elements, without losing all the possibilities of URL. A more general case of the latter is URI – Uniform Resource Identifier, which does not necessarily come down to an internet address, since it can be represented not by a URL, but by a URN – Uniform Resource Name, which is compiled as follows

URN:<namespace>:<name> (of course, without the space).

Thus, if you are not required to include the real address of an element (or the latter does not exist), instead of

"http://my.fake.com/individuals/object1"

it is more correct to use

"URN:individuals:object1".

In both cases, the full identifier of the element can be separated into its **actual name** and the part preceding it, called the **namespace**. This term is related to the grouping function of the latter. After all, both "http://my.fake.com/individuals/" and "URN:individuals:" can be interpreted as some directory which stores, in this case, individuals. Likewise, the bigger the knowledge base becomes, the bigger the role grouping its elements into namespaces takes on.

There can be one namespace for any element; there can be three (for classes, for predicates, for individuals); there can be more, dividing different types of individuals into different spaces. SWT do not regulate the overall number and degree of structuredness of the spaces, therefore this problem is solved by the developer themselves.

Due to the fact that namespaces usually contain more than one element (or one space is used in the whole knowledge base, which is not preferred but not forbidden), we observe a constant repetition of the namespace in a triplet. This means that when designating it with a short name, you can increase the readability of the knowledge base code and reduce its size.

If we designate the namespace "URN:individuals:" with the shortened "inds", we can use "inds:object1" instead of "URN:individuals:object1" in any mention of the elements. Such designations as “inds” are called **prefixes**, each of which denotes its own namespace.

At the same time, the prefixes themselves and their corresponding namespaces are entered into the knowledge base code by the developer, declaring them like representing variables.

This is the same as in the case of using namespaces in high-level languages. The spaces can either be created or joined to already existing ones. At the moment we are approaching the concept of Linked Data, since including existing namespaces into their own databases and spreading their own publicly available namespaces with the elements contained in them serves to create a universal network of interconnected elements of various knowledge bases, which together make up a single base.

In order to standardize at least the basic elements common to all storages, several publicly available namespaces were created with the most frequently seen and important classes and predicates. These are discussed below.

Thus, at the lowest level of SWT we need to answer the question about which of the proposed elements of the knowledge base will be resources and which will be literal. To the point, this is where the line between semantic knowledge bases and neural networks lies: the first involves an object-oriented approach, when, for example, a choice between class examples or operating their properties occurs, which is problematic to implement using neural networks.

Notations in semantic web technologies

When a method for describing each element separately is determined, the question arises about the methods for connecting resources with resources and resources with literals. In fact, here we move onto issues of writing the code of the knowledge base itself.

Issues of notation – these are problems with syntax, since in regards to SWT, notation is generally the name given to developed knowledge representation languages, each of which has its own syntax. Several of them have been developed:

–RDF/PHP,

–RDF/XML,

–N3/TURTLE,

–JSON-LD and some others.

We will take two triplets of the following form

roboticArm isUsedFor detailsMoving AND roboticArm hasPart endEffector

and discuss how they will look completed in each of the designated notations.

In the notation **RDF/PHP**:

```
array ( 'urn:test:roboticArm' =>
array ( 'urn:test:isUsedFor' =>
array ( 0 =>
array ( 'type' => 'uri', 'value' => 'urn:test:detailsMoving', ), ), 'urn:test:hasPart'=>
array ( 0 => array ( 'type' => 'uri', 'value' => 'urn:test:endEffector',),),),)
```

In the notation **RDF/XML**:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:ns0="urn:test:">
        <rdf:Description rdf:about="urn:test:roboticArm">
                <ns0:isUsedFor rdf:resource="urn:test:detailsMoving"/>
                <ns0:hasPart rdf:resource="urn:test:endEffector"/>
        </rdf:Description>
</rdf:RDF>
```

In the notation **JSON-LD**:

```
[{"@id":"urn:test:detailsMoving"},
{"@id":"urn:test:endEffector"},
{"@id":"urn:test:roboticArm","urn:test:isUsedFor":
[{"@id":"urn:test:detailsMoving"}],
"urn:test:hasPart":[{"@id":"urn:test:endEffector"}]}]
```

And in the notation **N3/TURTLE**:

```
@prefix test:<URN:test:>.
test:roboticArm test:isUsedFor test:detailsMoving;
test:hasPart test:endEffector.
```

Due to the comprehensibility and readability of the last notation, we will use this one from now.

TURTLE (terse RDF triple language) involves representing triplets in the form of a subject, predicate and object, simply separated by spaces. At the same time, there should be no spaces inside each of these elements. The only exception is string literals, which are enclosed in quotes, and in which the Cyrillic alphabet can be used and text can be written in natural language:

test:roboticArm rdfs:label “robotic arm”.

It should be mentioned again that the string literal “robotic arm” also does not have a prefix, which means that it is not included in any namespace, since it is not a resource and, therefore, does not have an identifier (URI) and cannot be the subject of any triplet.

All resources must have an identifier, in connection to which prefixes are generally used to shorten code. These prefixes appear as shown above:

@prefix test:<URN:test:>.
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.

When using a prefix, resources are described in the form

prefix:actual_name,

for example, test:roboticArm.

Here it is important to consider that, when interpreting the code of a knowledge base, each prefix along with the colon following it (in the example above – “test:”) will be replaced with content angle brackets (in this example – with “URN:test:”). In connection to this, it should be made sure that the content of the angle brackets when representing a prefix ends with the vertical bar. The role of the latter is usually filled by the symbols “/”, “:” or “#”. Therefore, if the prefix in the example above, *test,* were described as “@prefix test:<URN:test>.”, test:roboticArm would not be interpreted as <URN:test:roboticArm>, but as <URN:testroboticArm>, i.e., we would have the single name “testroboticArm”. Similarly, without the symbol “#” in angle brackets when declaring the prefix rdfs, the element rdfs:label would be understood as <http://www.w3.org/2000/01/rdf-schemalabel>.

It should be noted here that the use of prefixes is not a requirement of syntax, therefore the triplet

```
test:roboticArm rdfs:label “robotic arm”
```

can be written in the form

```
<URN:test:roboticArm> <http://www.w3.org/2000/01/rdf-schema#label> “robotic
arm”.
```

In the case of using an identifier without a prefix, it is strictly necessary to use angle brackets for each resource.

The next important moment is related to the punctuation part of the syntax TURTLE.

Each triplet should end in a period except for **shortened notes**. The latter means the possibility of using, firstly, instead of a construction of three strings with repeating subject and predicate

```
test:roboticArm test:isUsedFor test:detailsTranport.
test:roboticArm test:isUsedFor test:heavyThingsTransport.
test:roboticArm test:isUsedFor test:researchWork.
```

the expression

```
test:roboticArm    test:isUsedFor    test:detailsTranport,
                                     test:heavyThingsTransport,
                                     test:researchWork.
```

These are the same three triplets but written in shortened form. For this, the objects are separated by a comma, while a period is used once.

Secondly, the shortened note can also be used when repeating the same subject in several triplets, in connection to which a construction from, again, three triplets

```
test:roboticArm    test:isUsedFor    test:detailsTranport.
test:roboticArm    rdfs:label        “robotic arm”.
test:roboticArm    rdf:type          test:LabEquipment.
```

can be replaced by a simpler note

```
test:roboticArm    test:isUsedFor    test:detailsTranport;
```

```
rdfs:label        "robotic arm";
rdf:type          test:LabEquipment.
```

This kind of note is the most preferred: all triplets should be grouped (and preferably sorted) by subject using the symbol ";", as shown above.

If we remember about the blank nodes mentioned earlier, special syntactic means of expression are pointed out for them: they are not resources and therefore do not have an identifier. However, all of them can act as the subject in a triplet.

They are marked by [ ], but in this form, they are used only if the analyst want to convey that there is nothing to put in this place. In the case of using them for connections (as in the example with the place and time of a lecture), the predicates and objects, for which the blank node is the subject, are written directly inside it. In connection to this, the given example should be coded as:

```
test:lectureKE  test:takesPlace [  test:time    "14:00";
                                   test:room    310 ];
                test:takesPlace [  test:time    "16:00";
                                   test:room    314 ].
```

In other words, the subject of a blank node in square brackets is simply left out. At the same time, in each of the blank nodes (in square brackets) in our example, two triplets are immediately indicated implicitly, without a subject.

However, there is a possible situation where the knowledge base has several blank nodes which are necessary to distinguish from each other in order to be able to attach predicates and subjects not in one string, as in the example above, but in random sections of the code. **Blank node identifiers** exist specially for this case. Instead of a prefix, they always contain the symbol "_". In sum, the example described above with the introduction of blank node identifiers can take the following form:

```
test:lectureKE  test:takesPlace    _:id01;
                test:takesPlace    _:id02.
_:id01          test:time          "14:00";
                test:room          310.
_:id02          test:time          "16:00";
                test:room          314.
```

Next, having discussed the simplest methods of working with resources and literals, we need to move onto the next levels of the basic model of SWT.

RDF and RDFS

The concept of Linked Data itself involves not only the possibility of using publicly available namespaces with the elements that make them up, but also specific rules for standardization which are focused on being able to use rules common for all structures.

W3C as the developer of SWT introduced several publicly available namespaces, designed to provide all users with a single set of elements (classes and properties) which users will apply for solving specific problems without needing to develop their own.

As an example, we will look at the problem of denoting that a specific individual belongs to a class or classes. For this, it is possible to create one's own property test:belongs or something similar. However, if each user creates and inputs their own properties for this purpose, the use of anyone's knowledge base will first require solving the problem of searching for suitable properties for connecting individuals and classes. However, if this is done by a software agent, the problem is not at all obvious.

Instead of this, when developing their own bases, all users are suggested to use the namespace RDF (Resource Description Framework) and the property contained within it **rdf:type**, which connects its subject individual to a class, the object, for example:

```
test:roboticArm  rdf:type  test:labEquipment.
```

At the same time, if instead of "rdf:type" we write simply "a", the interpreters supporting SWT perceive "a" as a shortened form of "rdf:type", and therefore, an even shorter note can be used

```
test:roboticArm  a  test:labEquipment.
```

Of course, this implies the pre-declaration of the prefix rdf:

```
@prefix  rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
```

At the same time, this URL is functional, i.e., we can open the file describing all elements contained in this namespace and pick out among them the ones necessary for a project.

We will look at the most important elements for us of the namespace RDF.

The class **rdf:Statement** is used in cases when it is necessary to make an assertion about another assertion. For example, about the activation of a specific triplet in a knowledge base: if a certain sensor shows a temperature higher than the given threshold, information needs to be added to the knowledge base about how the system is reaching an unstable state. In this example, the triplet "system hasState nonStable" should be "inactive" before this moment and now to be activated. To separate these assertions from other examples of the class rdf:Statement, they can be designated as examples of the class State, which they in fact are semantically.

To distinguish such statuses, an example of the indicated class needs to be created:

```
test:nonStableState a rdf:Statement, test:State.
```

Next, we will attach to it the subject, predicate, and object of the triplet, which we want to be able to activate and deactivate using the properties **rdf:subject**, **rdf:predicate** and **rdf:object,** respectively as follows:

```
test:nonStableState   rdf:subject     test:system;
                      rdf:predicate   test:hasState;
                      rdf:object      test:nonStable;
                      test:status     test:Disabled.
```

Thus, the triplet about the state of a system turns out to be contained within the knowledge base, not as a triplet, but as three individual components bound to the example of class rdf:Statement. Further on, in certain conditions, it will be possible to transfer to the knowledge base control system lists of names (identifiers) of the example rdf:Statement which need to be activated/deactivated. Then, if a program handler had been previously developed or imported, it can find all activated examples of rdf:Statement in the knowledge base and add their components explicitly to the knowledge base, as well as find all unactivated ones and explicitly delete them from the knowledge base.

The next important element is the lists, which are sometimes called collections (RDF collection). They will be of importance when we discuss the problem of formalizing rules.

For now, we will look at the structure without semantic explanations:

```
        test:disClasses1  owl:members  ( test:Book  test:Car  test:Person ).
```

The elements in the parentheses, separated by a space, also make up the components of the list **rdf:List**. This note is automatically transformed into a rather interesting bundle:

```
test:disClasses1    owl:members      _:autos1.
_:autos1            rdf:first        test:Book;
                    rdf:rest         _:autos2.
_:autos2            rdf:first        test:Car;
                    rdf:rest         _:autos3.
_:autos3            rdf:first        test:Person;
                    rdf:rest         rdf:nil.
```

Anyone familiar with lists in high-level programming languages easily recognizes here the indicators from each previous element of the list to each subsequent one, more specifically: to each new sublist designated by the identifier of a blank node _:autos*N*. Using the elements **rdf:first** and **rdf:rest** does not require an explanation, while the individuals **rdf:nil** of class rdf:List are denoted by an empty list, which, logically, is similar to NULL as a final element.

The main reasons for why it is worth using lists is the simplicity of completing lists in the knowledge base code (simple listing in brackets) and the presence of special mechanisms for requesting elements according to their numbers, as well as the display of all elements of the list.

The last point which may interest us in this namespace are the classes rdf:Bag, rdf:Seq and rdf:Alt, which are subclasses for rdfs:Container. Here, no special mechanisms are implemented (unlike with rdf:List) for obtaining elements from the container, and the code for its creation is written differently:

```
test:bag1 a        rdf:Bag;
          rdf:_1   test:roboticArm;
          rdf:_2   test:quadCopter;
          rdf:_3   test:laserSensor.
```

At the same time, the syntax for all three types of containers is the same. The difference between them is semantic: **rdf:Bag** includes elements the order of which does not matter (for example, a list of available equipment); the order designated in **rdf:Seq** is important (for example, the order of actions in an emergency situation); the container **rdf:Alt** includes mutually exclusive elements (i.e., the very fact of including elements in this container has semantic value).

This is where the entire scope of the namespace of interest to us, denoted by the prefix rdf:, ends.

Despite the presence of a set of useful elements, they are not enough for standardizing certain important descriptions in knowledge bases. For example, we can indicate “test:roboticArm a test:labEquipment”, but the object of this triplet, the class test:labEquipment, does not take on any semantic meaning for software agents, since rdf: has no tools for establishing connections between classes, let alone tools for explicitly defining test:labEquipment as a class.

Due to the lack of standard tools for solving these (and other) problems, the following namespace was introduced. It is denoted by the prefix **rdfs:**, which means “RDF Schema”. In other words, we are talking about linking into a scheme, about building some system.

Thus, in the indicated namespace with the address http://www.w3.org/2000/01/rdf-schema# (which is also current and makes it possible to explore the content), we can find the property **rdfs:subClassOf**. The purpose of the latter is rather transparent. It is necessary only to clarify that its use for linking individuals and classes is not acceptable, since the property rdf:type (or abbreviated to “a”) is used for this. **rdfs:subPropertyOf** is used to denote the subproperties.

Here, of course, we must also mention the class **rdfs:Class**. If we need to indicate that test:labEquipment is a class, we should write

test:labEquipment a rdfs:Class.

In other words, any class is an individual of class rdfs:Class. This is one of the rare situations where the class itself is looked at as an individual. Equally, any predicate can be considered as an individual of the general class **rdf:Property** from the previous namespace.

The properties **rdfs:label** and **rdfs:comment** are also frequently used:

test:nonStableState	rdfs:label	“Non-stable state of system”;
	rdfs:comment	“Instability, action required”.

They help with the task of displaying materials from the knowledge base for users in an understandable form, since they might understand the structure test:nonStableState incorrectly.

The last pair of properties we are interested in is **rdfs:domain** and **rdfs:range**. At the same time, the subject in a triplet with any of these properties should be a property in the role of a predicate in another triplet. In other words, domain and range are properties of other properties and characterize them.

We will look at the triplet

test:lowBatteryLevel test:causes test:WirelessAgentDisconnection.

We can see in this example that the predicate expressed by the property test:causes has the subject test:lowBatteryLevel and the object test:WirelessAgentDisconnection. If we want to indicate that the *subjects* of the predicate expressed by the property test:causes should belong only to the class, for example, test:ProblemReason, we create the triplet

test:causes rdfs:domain test:ProblemReason.

Accordingly, in this case, the individual test:lowBatteryLevel (having the property test:causes) must be automatically assigned to the class test:ProblemReason.

The property rdfs:range works the same way, but it regulate the class which the *objects* of the predicate expressed by the described property should belong to, for example,

test:causes rdfs:range test:Problems.

The namespace connected to the prefix rdfs: also does not provide sufficient functionality for our tasks, therefore we need to consider the following. Before this, however, in order to be able to start working with the knowledge base materials, we will look at another tool – SPARQL.

SPARQL

We should remember the term mentioned earlier "knowledge base management systems" (KBMS). KBMS is a software module performing specific actions with the knowledge base, since the latter is itself only a set of files with code in a specific notation. In other words, the knowledge base itself cannot provide any functionality for working with knowledge, besides the direct storage of facts and rules.

KBMS must provide the following functionality for working with knowledge:

- request the knowledge from the base indicated by the user;
- input changes into the base indicated by the user (adding triplets, changing and deleting them);
- automatic display of new knowledge based on old knowledge (knowledge inference);
- automatic check of the knowledge base for inconsistencies (inconsistency checking);
- providing integration with other tools of storing and processing knowledge (for example, with a storage of mathematical formulas and a module for performing calculations).

Most of these functions are united by the need to request from the knowledge base the facts and rules necessary to complete them, with a subsequent processing of the query results. This explains the need for studying and implementing a mechanism for processing requests when working with knowledge bases.

In order to accomplish this in the framework of SWT, developers suggested a tool called SPARQL (recursively denoting SPARQL Protocol and RDF Query Language). As the name suggests, this tool includes several components at once:

- SPARQL Protocol;
- SPARQL language;
- SPARQL Query Results XML Format.

The need to introduce the protocol is justified by the concept of Linked Data, involving the possibility for different users to refer to any open knowledge base. However, since the latter are physically located on different servers (and in different countries), one of the simplest methods was to develop a protocol add-on of HTTP to refer to a server storing the knowledge base containing the needed information.

The software module applying and processing queries in SPARQL, sent there via the SPARQL protocol, is called **SPARQL endpoint**. Since SPARQL protocol works on top of HTTP, for a request from a remote node, it is necessary to send an HTTP request by IP address or, in general, by the URL address of the endpoint.

The general formula for transmitting a request in the form of an HTTP string looks as follows:

{URL endpoint}?query={request text in SPARQL}

In the case of the popular endpoint http://dbpedia.org/sparql, the string will accordingly look like this (for example):

http://dbpedia.org/sparql?query=select+distinct+%3FConcept+where+%7B%5B
%5D+a+%3FConcept%7D+LIMIT+100

Note that since, again, the designated protocol works by HTTP, following the given link already means the request has been sent. Therefore, by following it, the user will immediately see a table of results.

The table presentation of this result is also not a coincidence and not the idea of the developers of this endpoint: it is a formatting requirement for presenting results of SPARQL queries. To clarify, the table is the most convenient form for expressing the result: each bound variable is in a new column, and each result is in a separate row.

Of course, presenting results to users is not the main goal of the SPARQL endpoint, which is in large part needed by software agents to read this data. Here is where the uniformity of results is fundamentally important. Thanks to it, an agent can send a request to any active endpoint, and the result from any of them can be processed using the same algorithm.

The only thing left to understand is the language of the queries itself, which is inextricably connected to the term **triple pattern**, meaning an arbitrary triplet in which one, two or all three of the elements are replaced with **variables**. The latter are presented in the form of an arbitrary set of symbols, starting with the sign ?, for example: ?variable, ?newEquipment and so on.

The first thing to note is that, as in the programming language you are used to, the choice of the variable name does not impact its content in any way. It only impacts how comfortable it is for the developer to read. Its future content (value) is determined by the context of use in the request.

This happens in the following way: the user creates a query containing at least one triplet pattern. When processing the query, this pattern is superimposed onto all triplets in the base, selecting only those that fit the pattern. Triplets that fit the pattern are those whose elements match the constant (not variable) parts of the pattern. For example, the pattern "?equipmentInds rdf:type test:Equipment" is met by all the triplets in the base which have the predicate and object, respectively, rdf:type and test:Equipment. When a list is formed of all the triplets which correspond to the pattern, the query result (if no additional conditions are indicated) will be, in this case, a list of all the subjects of the triplets corresponding to the triplet (i.e., a list of resources which have the variable ?equipmentInds in their "place" in the triplet).

The full code of this request is:

```
PREFIX test:<urn:test:>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?equipmentInds
where
{
        ?equipmentInds rdf:type test:Equipment.
}
```

Note that this request is not to an external knowledge base but to a local one, with elements from the previously looked at examples. In connection to this, the namespaces and prefixes stated earlier should also be indicated, although the syntax differs slightly: the symbol "@" and the period at the end of the statement string are missing.

The keyword *select* needs to be followed by a list of all the variables whose values the requester wishes to obtain, while the request itself is formed after the word *where* in curly brackets. The request shown above leads to a list of possible values for the variable ?equipmentInds, i.e., the list of individuals of the class test:Equipment contained in this knowledge base.

The query

```
PREFIX test:<urn:test:>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?equipmentInds ?classes
where
{
        ?equipmentInds rdf:type ?classes.
}
```

will have an answer in the form of a table of two columns (since two variables are indicated after the word *select*), in which each string is an individual-class pair.

The next step for creating more flexible queries is to combine triplet patterns. At the same time, each of these patterns should be looked at as a set of results corresponding to it (as a variable value). Then the combination of triplets corresponds to the combination of result sets (during their overlapping, combination, subtraction, etc.).

The simplest operation here is **overlapping** results. Syntactically, this is described simply by adding another pattern to the query

```
...where
{
        ?equipmentInds          rdf:type        test:Equipment.
        ?equipmentInds          rdfs:label     ?indLabel.
}
```

Now, the final set of results show only those individuals which are at the same time described as individuals of the class test:Equipment and at the same time have an arbitrary inscription. Note that overlapping only works properly when there are general variables in “overlapping” patterns. Patterns intersected in this way form a **group of patterns**.

If we need to find in the result all individuals of the class test:Equipment which, conversely, have NO inscription, the structure MINUS {} is entered, subtracting the second set from the first:

```
...where
{
        ?equipmentInds rdf:type test:Equipment.
        MINUS {?equipmentInds rdfs:label ?indLabel.}
}
```

UNION {} is used in a similar way to combine sets.

Also of interest is the command OPTIONAL {}, which allows us to set optional conditions. If they are not met, the values of the variables will not be excluded because it is possible here to have empty cells in the table, i.e., in some strings, the variable turns out to be unbound to the value. This kind of situation is impossible without using this command, since in the opposite case, the result is considered to not satisfy the criteria, and the string is excluded completely.

Separately, we will talk about a request of all the elements of a list with the example

```
test:disClasses1  owl:members  (  test:Book  test:Car  test:Person  ).
```

For this, a query is sent in the following form

```
...where
{
        ?x          owl:members        ?list.
        ?list       rdf:rest*/rdf:first   ?item
}
```

The next important question is discussing the issue of filtering results. This can be completed using the keyword *FILTER*:

```
...where
{
        ?equipmentInds  rdf:type  test:Equipment.
        FILTER (<condition>).
}
```

At the same time, the filter always refers to the whole group of patterns because the string with this keyword can be positioned arbitrarily within the group.

The condition is often connected to comparing the variables to each other (?x = ? y), while checking correspondence with the resource or literal (?y = test:Class2 or ?z = 24 or ?y = “Problem”), with a numerical comparison (?z < 44).

Other keywords are also frequently found in the conditions.

For a symbolic comparison within FILTER, *regular expressions* is used, for example FILTER(*REGEX*(?x,<regular expression>,<flags>)).

To filter only the variable values which are present in the list (any of them), the keyword *IN* is used: FILTER(?x IN (test:Book, test:Car)). Negation: NOT IN.

In some cases, the function *bound*(?x) may be of interest, and in the negative form in combination with OPTIONAL {}. This function takes on the meaning “truth”, if the variable of the pattern has been assigned some value. So, the structure

```
...where
{
        ?equipmentInds rdf:type test:Equipment.
        OPTIONAL {?equipmentInds rdfs:label ?label }
        FILTER (!bound(?label)).
}
```

allows us to extract individuals of the class test:Equipment which *do not have* any inscription (in other words, the variable ?label is empty). Thus, we have a filter for absence.

The condition can be complex and can include the operations && (logical AND), || (logical OR) and “!” (logical NOT).

The last point which is touched on in this section is aggregate functions which perform calculations on the entire sample obtained upon processing the request. We will look at only one of these: count(?x), counting in general the number of triplet patterns satisfying the search conditions. Thus, the following request counts the number of individuals of the class test:Equipment.

```
select count(?equipmentInds ) as ?count
where
{
        ?equipmentInds rdf:type test:Equipment
}
```

Note that in this example, no variables are indicated between the words *select* and *count*. Including variables immediately changes the results of the calculations: with variables present between *select* and *count,* the processor computes how many times in the results each of the various values of the indicated variable is met. In this way, a query in the popular terminal http://dbpedia.org/sparql with the following form:

```
select ?author COUNT(?author) as ?count
where
{
        ?works dbo:author ?author.
}
```

counts how may works were written by each author shown in the results (technically: in how many strings of the results each author repeats in the pattern ?works dbo:author ?author).

Next, it is already possible to filter out people who are authors, for example, of at least four works.

```
select ?author COUNT(?author) as ?count
where
{
        ?works dbo:author ?author.
}
Group by ?author
having (count(?author) > 3)
```

All remaining information about using the query language SPARQL can be found on the website w3.org.

Using the library dotNetRDF

Everything mentioned above is sufficient to create a simple system for managing a knowledge base. At the same time, it is clear that the knowledge base alone cannot manage itself – it requires an application which will be installed in it for obtaining and changing knowledge. Accordingly, the management system should be able to read and write files of the knowledge base in one of the SWT notations. It should be able to send SPARQL queries and process their answers. It should be able to add/ change/delete triplets and their parts within the knowledge base.

In order to not implement the whole designated functionality manually, multiple pre-prepared libraries should be used, which exist for popular programming languages:

- for the developing language C# as part of dotNet – the library dotNetRDF;
- for Python – RDFLib;
- for Java – OWL API.

We will further discuss interaction with knowledge bases using the library dotNetRDF. The main principles can be transferred to other libraries and tools of development.

The main collective term here is **knowledge base graph**. A graph is a set of peaks and pointed edges. At the same time, here, unlike with the concept map, the **nodes** of the graph are considered to be the individuals and classes, and even the properties themselves.

All triplets of a knowledge base make up a graph, therefore work with it is carried out using the class *Graph*. Actions with the knowledge base are actions with an example of the indicated class which needs to first be initialized:

Graph kb = new Graph().

The knowledge base graph can be completed either by reading from files or by manually adding/programming individual triplets into the graph.

In the first case, it is enough to create an individual from the class *Notation3Parser* (when using the notation TURTLE) and then using it with the *Load* method, indicating the address of the read file, for example:

```
Notation3Parser parser = new Notation3Parser();
parser.Load(kb, @"D:\Ontology\KB\kb.n3");
```

In the second case, we need to form individual triplets and then add them into the graph. In this case, the node of the subject, predicate and object of each triplet is created separately (or is retrieved from the knowledge base using a query). It should also be taken into account that there are **URI-nodes**, and there are **literal nodes**. When working with both types, the interface *Inode* is used:

```
INode subj = kb.CreateUriNode(“test:equipment”);
INode pred = kb.CreateUriNode(“test:hasMass”);
INode obj = kb.CreateLiteralNode(500);
```

Next, all three elements are added into the base with one triplet:

```
kb.Assert(subj, pred, obj);
```

They can be pre-grouped into a triplet:

```
Triple trip1 = new Triple(subj, pred, obj);
```

which can be added with the same method: kb.Assert(trip1).

To delete a triplet from the base, the *Retract* method can be used in a similar way.

At any moment of time, all triplets of the knowledge base can be seen using the *Triples* properties, and the result can be transformed into a list using the ToList() method:

```
kb.Triples.ToList(),
```

then each of its elements will be an example of the already familiar class Triple, therefore we can retrieve an element from the list:

```
Triple trip2 =kb.Triples.ToList()[2].
```

Next, using the properties Subject, Predicate and Object, we can obtain the component elements of the triplet, each of which is compatible with *INode* and, therefore, can be used for forming and adding new triplets. Note that there are two ways of working with nodes: by creating them using CreateURINode/CreateLiteralNode or using preexisting ones, for example:

```
INode pred = trip1.Predicate;
```

It is also possible to use nodes retrieved from the query, therefore the next important task is to ensure a query is sent to the knowledge base, which also involves processing its answer. To send the request, the *ExecuteQuery* method of class Graph is used, whose input is the *string* of the request itself. To process the result, an example of *SparqlResultSet* is created, and the result of the query is explicitly brought to it:

```
SparqlResultSet rezSet = (SparqlResultSet) kb.ExecuteQuery(queryString)
```

In this example, rezSet is the list of the individual results *SparqlResult,* therefore we can use the index to access a specific result of the query, for example,

```
SparqlResult rez = rezSet[5].
```

At the same time, we must remember that rezSet is the table of results which we see when processing a query, for example, on the terminal dbpedia.org/sparql. Then rez is one of the strings of the table, the number of elements of which (as with the number of columns in the rezSet table) depends exclusively on the number of variables appearing in the request after the keyword *select*. Thus, rez is also a list, but one of already specific variable values.

Table 1 shows two strings from the results of the following query

```
select distinct ?ind ?indLabel
where
{
        ?ind a ?Concept.
        ?ind rdfs:label ?indLabel.
} LIMIT 100
```

To obtain the value of the inscription (ind) from the second row of Table 1, we need to apply it (taking into account numeration from zero) using rezSet[1][0] or rezSet[1][“ind”].

Table 1. Part of query results

Ind	indLabel
http://dbpedia.org/resource/Innovation_economics	“Innovation economics”@en
http://dbpedia.org/resource/International_Society_on_General_Relativity_and_Gravitation	“International Society on General Relativity and Gravitation”@en

It is very important at the same time to keep in mind that the results of a query can be used for creating triplets, for example,

INode pred = rezSet[1][“ind”],

and they can also be used to process new requests, for example,

```
string dynamicQuery = @“
select ?pred ?obj
where
{
        <” + rezSet[1][“ind”] + @“> ?pred ?obj.
} LIMIT 100”;
```

Note that this string of the request is dynamically formed using the results of the previous request. Furthermore, in the strings of the request (as in the code of the databases themselves in TURTLE), when transmitting the complete identifier (i.e., without a prefix, the address itself) created with a program or taken from, for example, the query results, triangular brackets should be used: <urn:test:equipment> or <{ rezSet[1][“ind”] }>, where { rezSet[1][“ind”] } is the value of the variable rezSet[1][“ind”] (of the Node type).

Next, we can move onto the following level of SWT, which we will later on apply the previously described software toolkit to.

OWL

The previous level of the basic model SWT has a main disadvantage related to solving the problem of forming knowledge (checking for integrity and obtaining new knowledge). More specifically, it is these problems that require more tools in the RDFS namespace. The only elements from them which can be used for creating rules and may in the future be applicable to the formation of knowledge are: rdfs:domain, rdfs:range, rdfs:subClassOf, rdfs:subPropertyOf. At the same time, the “class-class” type rules discussed above can be well described with the structure

test:A rdfs:subClassOf test:B.

However, the remaining types of rules require other tools of expression for their description.

To work with them, we need a higher level of the basic model SWT – the level of the namespace Web Ontology Language (OWL), whose physical address can also be found in any of the publicly available lists. If we look inside it, we can see that it contains several elements.

An important clarification needs to be made right away: on its own, the language OWL is only a common method, a standardized form of representing rules. OWL does not provide processors with these rules. It has no ready-to-use software tools for forming knowledge. Another thing is that, if so wished, one can find publicly available packages which provide some processing. However, these are independent products.

In order to get a deeper understanding of the process of the work of an inference machine, we will develop such products ourselves, which also allows us to develop on a “turnkey” basis. Therefore, it is necessary to study the elements of the namespace OWL and understand how they can be used.

The first of these, which we will start with, is the class **owl:Class**, which is a subclass of the familiar rdfs:Class, but which implies that these classes will be applied for logical operations, which should be completed within logical processors (in inference machines) and should serve the goals of forming knowledge: solving two of its known subproblems.

According to these subproblems, we separate the elements into categories, which will be looked at separately.

OWL-elements for discovering inconsistencies

Inconsistencies in knowledge bases can be expressed in the presence of one or several facts contradicting one of the rules written in the base (breaking the rules). Accordingly, to identify inconsistencies, it is first necessary to create and describe all the rules which regulate the impossible combinations of content of the knowledge base.

The first element in the list of those used for creating such rules can be called the property **owl:oneOf**, which imposes a limit on the possible examples of a specific class, indicating their final list:

```
test:CurrentlyUsed     a             owl:Class;
                       owl:oneOf   ( test:Arm2  test:MCU6  ).
```

Accordingly, using the rules with owl:oneOf to identify inconsistencies in the knowledge base, the system should first request all classes described using this element (?x owl:oneOf ?y), and then check the individuals belonging to the obtained classes for the presence of anything not included in the limited list presented in the rule.

The next element is **owl:disjointWith**, which allows us to immediately indicate a pair of classes, to which no individual can belong to at the same time, in other words – an inconsistency:

```
test:Car owl:disjointWith test:Book.
```

However, this way is not convenient to immediately indicate a whole list of mutually disjoint classes, therefore, in order to avoid prescribing many pairs, it is enough to indicate the list using the element **owl:AllDisjointClasses**.

```
[ ]     a                  owl:AllDisjointClasses;
        owl:members        ( test:Book  test:Car  test:Person ).
```

Accordingly, if there are several such lists, an individual or a named blank node needs to be used instead of an empty node.

There is also the concept of disjoint properties, which is described using **owl:propertyDisjointWith**. The semantic meaning is that two individuals cannot be connected to each other at the same time by several of the listed disjoint properties:

```
test:hasSon owl:propertyDisjointWith test:hasDaughter.
```

Here, we need to mention indicating the types of properties which {indications} on their own are already rules. For example, if a property is irreflexive (**owl:IrreflexiveProperty**), no individual can be connected to itself by this property (this is already a rule). However, if a property is asymmetric (**owl:AsymmetricProperty**), no two individuals can be connected to each other by this property "in both directions" (in other words, it is not possible for the simultaneous fulfillment of aRb and bRa).

There is also the class **owl:NegativePropertyAssertion** for the flexible denotation of the unacceptable joining of specific elements into a subject-predicate-object triplet, for example:

```
_:np1   a                       owl:NegativePropertyAssertion;
        owl:sourceIndividual    test:Arm1;
        owl:assertionProperty   test:inOrderTo;
        owl:targetIndividual    test:heavyMassTransport.
```

For a similar use of a literal property (unlike an objective one), owl:targetValue is used (instead of owl:targetIndividual).

As with the case of rdf:Statement, such triplets can be presented by one node (in this example _:np1) and can be activated or deactivated.

OWL-elements for obtaining new knowledge

It is easy to imagine how the already familiar elements rdfs:domain and rdfs:range can be used to obtain new assertions: if a property is found in the knowledge base which indicates the domain or range, we find all triplets where it acts as the predicate and conclude a new assertion about the fact that all of its subjects, accordingly, or objects belong to the indicated classes.

This logic should also work for an inference machine in terms of the elements of the namespace OWL: first we search for a rule, then the background assertion (one or several), and based on them, we create a conclusion assertion.

Useful properties for this task are those such as **owl:SymmetricProperty** and **owl:TransitiveProperty**. If the property R is symmetrical, the fact of the symmetry and connection of *a*R*b* gives us the conclusion assertion *b*R*a*. The transitivity of the property R and the presence of the connections *a*R*b*, *b*R*c* give the conclusion *a*R*c*.

The property **owl:inverseOf** works according to the same logic. It can be used to indicate that two properties are mutually inverse. The inverseness of the properties R and S and the assertion *a*R*b* gives us the conclusion assertion *b*S*a,* and vice versa.

To work with the considered composition of properties, the element owl:propertyChainAxiom is used, showing the composition of which properties this property is:

```
test:grandParent     rdf:type                    owl:ObjectProperty;
                     owl:propertyChainAxiom      ( test:parent  test:parent ).
```

To describe the classes, **owl:unionOf** / **owl:intersectionOf** is often used. They are used in order to express the following axiom: if an individual belongs to any one/all classes from the list, it also belongs to the indicated class, for example:

```
test:FragileEquipment  owl:unionOf  (  test:Sensor  test:Microcontroller  ).
```

Here it is critically important that the list is a prerequisite, and the class indicated as a subject is the conclusion, but not the other way around.

To indicate a pair of complementing classes, the property **owl:complementOf** is used as the predicate (in logic “if an individual does not belong to the first class, it belongs to the second”).

To indicate the equivalence of classes to each other (or to connect class and *restrict* them), **owl:equivalentClass** can be used. This is especially important in the case of connecting knowledge bases in which the resource identifiers are presented in the form of code, like in the example

test:Robot owl:equivalentClass <https://www.wikidata.org/wiki/Q11012>.

Similarly, but in relation to individuals, the property **owl:sameAs** is used.

However, all of the rules looked at are all rather simple in form and do not allow us to form more complex structures. In connection to this, we need to further look at the possibility of using OWL for describing the rules of the already discussed description logic.

Describing description logic

The modification of OWL 2 includes all the elements necessary for describing description logic.

To denote an empty class ⊥, owl:Nothing is used, and to denote the class-Universum T, owl:Thing is used.

The remaining elements are related to the concept of *restriction* characteristic for description logic, which can be defined as a set of individuals satisfying some condition. The latter is always related to the properties of an individual acting as the predicate.

Remember what terminological axiom is and which two types exist. At this stage, it is important to understand the following: in the expressions $C \equiv D$ and $C \subseteq D$, C and D can be filled by classes (i.e., their specific URI), as well as by restrictions, which are usually written in the form of blank nodes, although they can be presented as named individuals of the class **owl:Restriction** with their own resource identifier.

Restrictions are always related to the values of the properties, and therefore, the restrictions are classified according to the nature of the conditions imposed on these values.

The simplest restrictions are formed as such: “a set of all individuals whose value of property R is equal to …”. The value of property R (this is the object for the predicate expressed by the property R) within description logic is often called **R-**

successor. For this reason, this restriction can be transformed as “a set of all individuals whose R-successor is…”.

We will look at an example of a rule (terminological axiom) containing this type of restriction:

```
test:RoboticArm a                   owl:Class;
      owl:equivalentClass    [      a                 owl:Restriction;
                                    owl:onProperty    test:hasPart;
                                    owl:hasValue      test:endEffector ].
```

It belongs to the type $C \equiv D$, where D is a restriction described, in this case, as a blank node. The use of **owl:hasValue** is key here: this sets the nature of the restriction, while the property **owl:onProperty** is used in all restrictions.

It is also important to consider that in the case of equivalent classes (owl:equivalentClass), the rule should work in both ways, which means, in this case, two rules at once:

1. any individual whose successor for the property test:hasPart is test:endEffector, belongs to class test:RoboticArm;
2. if an individual belongs to the class test:RoboticArm, it must have the successor test:endEffector for the property test:hasPart.

Both of these can act as rules for obtaining new knowledge and as rules for checking for inconsistencies.

In the case of using rdfs:subClassOf (which corresponds to the operation of inclusion), no ambiguity occurs, and one rule is given: the smaller (included) one is covered by the description of the greater (including) one. The reverse is not true.

To describe *universal restrictions,* the element **owl:allValuesFrom** is used:

```
test:RoboticArm           a                  owl:Class;
        rdfs:subClassOf [ a                  owl:Restriction;
                          owl:onProperty     test:hasPart;
                          owl:allValuesFrom  test:RoboticComponent ].
```

Again we note that the restriction can act as the object and as the subject of a terminological axiom, or in both roles at the same time in one triplet.

To describe *existential restrictions,* the element **owl:someValuesFrom** is used:

```
[       a                       owl:Restriction;
        owl:onProperty          test:madeByWorker;
        owl:someValuesFrom      test:BestWorkers]
                                        rdfs:subClassOf    test:HighQualityProduction.
```

No special commentary is needed here, except for the fact that if an existential restriction acts as the <u>object</u> of a terminological axiom when using rdfs:subClassOf, this rule can only be used to check for inconsistencies.

OWL also lets us describe the *cardinality restrictions* discussed earlier. The only difference is in the use of one of the elements: **owl:cardinality**, **owl:minCardinality** or **owl:maxCardinality** to indicate the exact value of the number of R-successors of any class:

```
  [   a                       owl:Restriction;
      owl:onProperty          . . . ;
      owl:minCardinality      2                   ]       rdfs:subClassOf   . . .
```

In the case where it is necessary to describe a restriction like "a set of individuals which have {n} R-successors belonging to the specific class C" (i.e., Qualified restriction), **owl:onClass** should be used for establishing the class, as well as **owl:qualifiedCardinality**, **owl:minQualifiedCardinality** or **owl:maxQualifiedCardinality** instead of those looked at in the previous paragraph, respectively:

```
  [   a                               owl:Restriction;
      owl:onClass                     . . . ;
      owl:onProperty                  . . . ;
      owl: minQualifiedCardinality    5                   ]       rdfs:subClassOf   . . .
```

With this, the main approaches to formalizing knowledge can be considered to have been discussed.

PART 4. KNOWLEDGE-BASED SYSTEMS

Dynamic rules

However, the true intelligence of systems does not appear until they begin to not simply use rules to deduce new facts (or check existing ones), but start to apply rules to deduce or delete other rules. It is not simply about rules for working with facts, but about rules for working with rules for working with facts.

The main approach which we will use is the activation/deactivation of rules. The idea here is that the latter will be stored in a knowledge base but in a form which allows the status of the whole rule (active or not) to be indicated explicitly. Consequently, we need a form of note in which the rule completely "reduces" into a single node.

We have already come across this type of note, when we discussed the namespace rdf and its class Statement, in whose example a whole triplet is stored at once. It is, as it were, in the knowledge base, but at the same time, it differs from all the remaining triplets in its note form.

Thus, we can take any rule looked at in the previous section (always presented as a triplet!) and transform it into a special form, for example:

```
test:rule5    a                 test:Rule, rdf:Statement;
              rdf:subject       test:Car;
              rdf:predicate     owl:disjointWith;
              rdf:object        test:Book;
              test:ruleStatus   “deactivated”.
```

A restriction can be located in place of any one or two components of the rule.

Next, we need to work out the principles for activating the rules. The latter can be activated or deactivated when an event occurs. The following can take the role of an event:

- receiving a signal from a sensor/information system or receiving a command from a special agent;

– computing specific results of comparing certain values to each other;
– activating one or several rules or facts (for example, the activated rdf:Statement examples obtained as a list).

The first case implies that the information system does not give out the number itself (for example, the value of a specific sensor or the output of a neural network), but the assessment obtained from analytical processing in the form of a string, for example: “Sharp rise in pressure”.

Then the rule itself for activating the rule is put in the form

```
[   a                   test:RuleActivationReason;
    test:keyText        “Sharp rise in pressure”;
    test:activatesRule  test:rule5     ].
```

Now to activate all rules according to the rules of this pattern, we need to implement a handler, which sends a request like “which rules are activated by a sharp rise in pressure?”, and all of the rules found become “Activated”.

If the activator of a rule is some individual condition (for example, the activation of a specific state (as in the example with the class test:State discussed earlier) or the activation of a specific rule (the latter being in an “Activated” state)), a simple triplet like the following can be used

. . . activatesRule . . . .

Its subject is the prerequisite for activation, and the object is the activated element.

In the case of a combination of conditions, instead of a blank node, we need to use, for example, a collection in which each element is represented by a rule or by a current state, expressed by an activated example of the class rdf:Statement (as in the previously looked at example with the class test:State):

(. . . . .) a test:RuleActivationReason.

It is implied that all elements of the collection subject are activated – this is a necessary prerequisite.

To implement other methods of studying, additional tools are needed, which are looked at later on. First, however, it is necessary to note that studying a system is only one means of solving a particular class of problems, specifically: problems of uncertainty. This concept will be looked at next.

Uncertainty

Intelligent systems are unique in that they are capable of solving problem with considerable **uncertainty**. The latter, of course means some lack of data or knowledge for further actions.

However, it is necessary to clearly understand that, for example, if we do not know in advance which temperature values will be given by a corresponding sensor, this in itself is not considered to be uncertainty. If the system receives clear sensor readings, and there is knowledge about which of the algorithms in the system should be used for all possible temperature ranges, then nothing at all interferes with the system making decisions. Therefore, this situation has no relation to uncertainty.

If, however, in this example, we want the system to continue working properly even in the situation where the sensors for some uncertain reason at one moment stop providing information, we are dealing with the **uncertainty of the situation**, characterized by insufficient information about the current state of affairs (there is no information from the information-measuring system or this information is unreadable); in other words, even if there is a suitable algorithm on board the system, there is nevertheless still not enough input parameters (since they are unknown) to launch it and begin actions. Another example is the situation with discovering an unidentified object which the system nevertheless has to make a decision about.

At the same time as all of this, however, an intelligent system must not stop working with an error or notification about a lack of data (although, a notification should, of course, be sent). On the contrary, it must in this case take certain special actions: either obtain this data some other way / from another sensor, or obtain other (compensating) data, or make a decision without some of the data. However, this decision must not break the code of the whole process or lead to undesirable consequences. Of course, if in all ambiguous situations the same decision is made (if the system is programmed for this action), this points to a lack of any intelligence. An intelligent system should analyze its own experience or that of another, extract patterns from it, perform the safest test actions, analyze their results and errors, etc.

It is important to understand that an intelligent system should create its own algorithm (at least part of it) during its work.

The next type is **uncertainty in action algorithms**. This has to do with the lack of a ready solution for implementing the purpose of the system, even if all the necessary input data is given. For example, if it is known that at any moment any command (in a form understandable for the system) can be given by a user or a higher agent (programs or devices), any required data about the situation is available, but the system may not have a pre-prepared method for executing the given command (the risk of lacking a ready algorithm grows with an increasing level of the arbitrariness of its content). Another example: a vehicle control system discovered some object on the road. All of its characteristics are known; however, its parameters do not correspond to the expectations of the system (for example, aggressive behavior). This includes the situation with an unpredictable change in the very purpose of the system.

Here, again, the system has no right to respond with "error" and stop: it is obligated to find a solution, taking specific actions for this which are suitable for this specific situation.

The third case of uncertainty is **uncertainty in the outcome**. This means that even if it is known which algorithm should be applied, the results of this application may not be the ones the system is expecting. This can be about a change in the executive or another equipment or a change of the plant.

At the same time, as it was said before, an intelligent system must be ready to process any outcome that occurs from its actions and, in the case of a discrepancy between the results and expectations, to adjust its methods and/or algorithms.

In fact, this division is not strict, since the given types of uncertainty are interconnected, often one following the other.

Next, we will look at the basic system device, potentially capable of resolving the specified uncertainties.

Components of a knowledge-based system

It is understood that in order to have intelligence, a software package cannot be made up exclusively of a semantic knowledge base and a system for managing it. This is due to the lack of a mathematical apparatus in SWT, which noticeably complicates the storing and processing of mathematical dependencies and even performing simple computing operations with incoming data. A knowledge base can, for example, contain information about what needs to be done in the case of a drastic change in the monitored parameters, but SWT cannot help track the very fact of this change. Furthermore, we would be deprived of the most important source of new rules and facts – the ability to analyze existing or new experience, archived and real-time data about the operation of the system.

Thus, an intelligent knowledge-based system (KBS) must include not only a system for managing knowledge bases, but other components as well (Fig. 10).

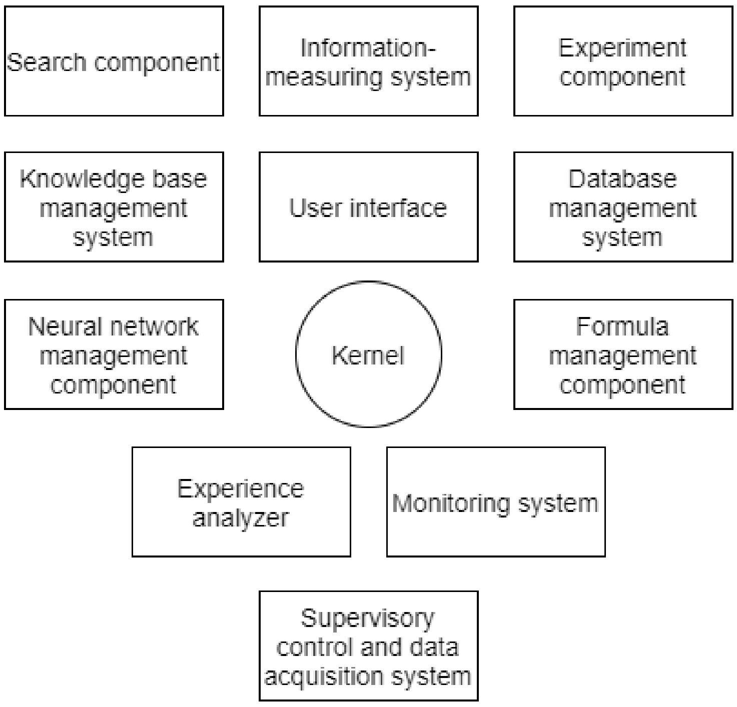

Search component
Information-measuring system
Experiment component
Knowledge base management system
User interface
Database management system
Neural network management component
Kernel
Formula management component
Experience analyzer
Monitoring system
Supervisory control and data acquisition system

Fig. 10. Components of a knowledge-based system.

We will look at the components separately.

Information-measuring system

First, it should be mentioned that the source of all real-time data is an information measuring system. This term will be understood as the totality of all sensors, all installed devices and agents from which information can be received. In other words, any data about the external environment enters a knowledge-based system through an information-measuring system.

The output point of this system we can consider to be the totality of files into which the system writes real-time data. In this case, the remaining components can refer to such files with read access to obtain data from them and then send it, for example, to a database or a monitoring system.

Formulae management component

Its purpose is to store mathematical (including logical) dependencies, as well as to call for their calculation.

The main difference of this component is, first, the storage of random analytical dependencies, where the latter can be modified directly during completion of the main system process, and, second, the ability to call for the required function by its unique identifier (also stored in the knowledge base).

The first feature is explained by the fact that an intelligent system offers the opportunity to learn – including the possibility of refining mathematical models during the work. The second is connected to the possibility of using various formulas to compute the same parameters depending on the situation. At the same time, since we are talking about learning, it is not known in advance, in which of the situations any formula will be applied.

To practically implement the specified possibilities, we can use, for example, dynamic programming languages (like Python, in which files with codes are interpreted "on the fly"), which should be used for describing mathematical functions for calculations. In the case of using C# language for developing this module in the Visual Studio environment, we used IronPython, which allowed us to call functions in Python from the main code in C#, even if the former were modified only after launching the main program.

This tool helps implement the second feature, since it allows us to call functions even with a predetermined name but from a random file, the name of which is indicated using a variable.

Accordingly, in the knowledge base, a function identifier is described for each formula. This identifies the function where the formula is implemented and the conditions of its use (for example, to calculate what value the formula requires and which input parameters are needed for this). For this, the properties “identifier”, “output value” and “input parameter” can be used for individuals of the class Formula.

For the direct application of an arbitrary formula, the *formulae management component* should have a provided function whose input parameters will be all the specified properties, and which will use, for example, again, IronPython to perform calculations, and which will return the received number or its qualitative estimate, e.g., “Exceeding permissible value”.

Since this component must provide the opportunity for prompt changes to formulas, it is necessary to solve the issue of its technical implementation.

An important point here is the following: unlike the components of an experience analyzer and neural network controller, this component is not designed to perform procedures of identification, approximation, interpolation and so on. Instead of this, this component requests already created formulas in various ways:

- asking an expert – via user interface;
- importing file projects;
- importing from third-party CAD-programs using the standard Component Object Model (COM) as shown in [Onufriev et al, 2019].

In the case of importing, special interface modules need to be developed, whose task it is to store formulas in a form suitable for use by this component.

Neural network management component

When solving problems, a person, acting as an intelligent agent, uses their ability to recognize objects around them and in images, the shapes of lines on graphs, patterns of element positions, etc. Since an intelligent system must also often solve these problems, and more often than not neural networks are used for this, intelligent systems should also include a **neural network management component**.

Remembering that neural networks are capable of solving a number of tasks, including making decisions about the behavior of complex systems, it is recommended all the same in the framework of knowledge-based systems to delegate to neural network, on the contrary, the simplest problems, such as:

- discovering the presence (or lack) of a specific object in a video stream;
- determining the fact of a quick (or slow) growth (or decrease) of a particular signal;
- detecting a drop of signal values (or group of signals) to a certain range(s) of values;
- determining the signal (or combination of signals) of a particular form: sharp peak, parabola, maxima and minima, etc.

In all of the specified cases, we are talking only about detection, without any decisions being made, since the more functions that are required from a specific neural network, and the more it solves at one time, the more complicated its structure and training is, and the greater the possibility of error in its work and set up.

In connection to this, the *neural network management* must contain a set of the simplest neural networks, which are only the source of preprocessed data for subsequent decision making by other components.

It is worth considering that some networks learn during their work (and in this case, this component is responsible for the direct execution of learning algorithms), while others are formed according to the type of knowledge-based neural network (KBNN, KBANN).

Monitoring system

A monitoring system is used to track various types of changes occurring with the flow of real-time data. This system can simultaneously use neural networks, formulas and other means of computing. The result should be sent to the knowledge base management system for subsequent processing.

However, neural networks, like formulas, by default return numerical values on the output. If all of this data is immediately sent to KBMS, uncontrolled, we end up with a number of requests which, in fact, may be repetitive and/or requiring no action. In this case, the monitoring system should send requests to KBMS only in situations of discovering specific signs. Likewise, it should determine the frequency of these requests.

On the other hand, a SPARQL-query to a knowledge base that includes a comparison with a numerical value, which is bound in the knowledge base to a specific indicator, has a greater length and will take longer to perform than a query to a state denoted by a string. At the same time, the structure of this query is relatively simple and is described above.

The monitoring system should be in charge of not only processing the results of a mathematical data analysis, but also for creating the processing tools themselves. At the same time, the role of the latter can be filled not only by neural networks but also by any algorithms for processing information, whereby these algorithms must be programmatically derived from the information stored in the knowledge base. This can be done, for example, by substituting the critical values requested from the knowledge base into the patterns of Python-function or into files in which neural network coefficients are stored.

The monitoring system uses neural networks and/or formulas for its tasks, from which comes the need to carry out interaction between the corresponding components.

Furthermore, the monitoring system supplies another component – an experience analyzer – with information about deviations of some indicators from the norm, which, in fact, prescribes functionality to it for tracking the need to launch an adjustment to the existing models.

Databases and knowledge bases management systems

The next component is a **database management system**. The latter are used to store the current indicators of a controlled process, which then can be used for analysis. Here we are dealing with not only technical indicators of a lower level like temperature, pressure, and concentration of gases, but also about performance indicators of a higher level, from the efficiency of a conventional "workshop" to the strategic indicators of the whole company or industry. DBMS should provide the opportunity to promptly write new data in conditions where it is not understood in advance how many data fields (columns) need to be stored. This number can be a variable.

Knowledge bases are not used for archiving information. Instead, they contain:

- control goals and their indicators;
- information about the sequence of necessary actions in response to qualitative assessments of situations coming from neural networks and formulas (like "Sharp rise in pressure");
- information about which of the formulas/neural networks (identifier of function processor) is needed for calculating which parameters, and which input parameters are needed for this;
- information obtained from experience, for example, about which actions in what conditions most often led to the correct result;
- information about controlled processes, allocated time rates, subprocesses and about their sequence, about their performance indicators and about the possible executors of each of them;
- information about agents (and workers, equipment), which can be involved in the processes, about their condition, about their interchangeability, about the criteria for selecting an executor;

- information about the most possible agent performance (for distributing the load);
- the sequence of actions during interaction between agents as part of solving general or related problems;
- rules for logically deducing or checking for inconsistencies.

Thus, a **knowledge base management system** stores and generates information about what and in what sequence needs to be done in order to meet the goals.

Presented next is an incomplete list of the material obtained from the DBMS output when interacting with it:

- list of control actions which need to be completed to improve the performance indicators;
- list of assessments of the state of the system as a whole;
- list of suggested reasons/consequences of this situation;
- identifier of the formula/neural network for calculating a specific value;
- list of executors which can be focused on solving problems as well as the compliance of their resources with the requirements put forward, and so on.

Here it is important to take into account interaction with formulas and neural networks: a knowledge base management system should take from them generated messages about the assessment of a situation, which means storing in it a list of such messages in order to be able to sent a SPARQL-query containing the appropriate information.

Furthermore, in the case of tracking the deviation of performance indicators from the established norm, a mathematical apparatus must be created automatically based on the settings stored in the knowledge base. If program methods are used to compare values, they should make comparisons with the settings obtained from DBMS. If neural networks are used, they do not require studying. This is a question of calculating "weights" according to easily derived formulas based on the required settings.

Experience analyzer component

If a system collects enough data about work, we can speak about the work of another module: the **experience analyzer component**. First of all, we are talking about detecting some patterns in the accumulated data, about defining connections (mathematical models) between various indicators. To solve such problems, this component should involve a statistical apparatus. Likewise, one should study their own neural networks, and other Data Mining technologies should be used.

When implementing this component, the following questions need to be answered:

- which event can become a sufficient basis for launching an experience analysis?
- which data specifically from all available data needs to be analyzed first?
- what does it need to start being analyzed for?

In order to simultaneously solve the given questions, it is suggested to use the monitoring system mentioned before, which is intended for tracking the state of signals and, in this case, can interact with the experience analyzer and initiate the process of searching for new knowledge based on the obtained data.

The first indicator which this system should react to is the deviation of one or several performance indicators from the previously given (or previously calculated) value. In general, this *can* mean (and therefore requires checking) that the general model which we used to calculate the required control action for ensuring the desired indicator does not work correctly. Note that it is not necessarily an incorrect *mathematical* dependency. After all, we can also talk about establishing an incorrect cause-and-effect relation between the performed actions and the expected results. A more general case of launching the analyzer is obtaining an unexpected result at the output (even when it is positive).

The experimental results between the desired results (indicators) and input interactions should be analyzed on the considered basis for starting the process of calculating models.

The second general basis for starting calculations is the fact of discovering a certain repetition. In many ways, this corresponds to the logic of reinforcement learning (which is inherently training), where after several repetitions, the same actions lead to the same result, which lets us create the rule "if A is done, the result can be B". To carry out this type of analysis, the database should contain, among other things, information about actions and about their outcomes.

Thus, the monitoring system should regularly refer to the database and, constantly processing the flow of real-time data written in it, track the state of key indicators and periodically monitor the repeatability of cause-and-effect relations.

When discovering deviations or repeatability, the experience analyzer component should be activated, already directly performing calculations. In the case of a deviation of the indicators, it should implement identification algorithms (including neural networks), and in the case of the repeatability mentioned – algorithms of statistical analysis.

It is worth noting that neural networks are used within knowledge based systems in two modes:

1) the neural network constantly checks the incoming flow of real-time or archived data;
2) the neural network performs an approximation of data for calculating a new model when one of the previously indicated events occurs.

In the first case, the neural network must be prepared for discovering the required/ undesired elements or patterns before being used. It is also possible to "re-train" it during the work. This mode can be used for monitoring the situation at a management facility to monitor the current state.

In the second case, the models are obtained by launching a process of training the network on the available data. It is important that both starting training and achieving its results should be controlled by the system itself with minimal human involvement. Thus, here the problem of prompt, unsupervised training arises.

We will add separately that the basis for starting the process of analyzing experience can also be served by the signal of the experiment component (looked at below), conducting the experiment and generating new experience for analysis.

User interface

This component is necessary for interacting with the user of a system. At the same time, it should be noted that a knowledge-based system may not have a user at all, since it can be designed from the beginning as completely autonomous.

The user interface serves for obtaining data from the user or for visualizing the current situation for the operator/person responsible.

For the input from the user, this component can take:

– commands to perform;
– knowledge;
– assertions in natural language (for example, in dialog systems);
– files (including graphs) for processing.

For the output for a user, the interface can give:

– warnings, notifications;
– recommendations;
– numerical statistics;
– graphs and diagrams visualizing the process;
– knowledge in a structured form.

In the case of working with technological processes, this component can act as the so-called human-machine interface (HMI).

Automated dispatch control system

If KBS fulfill not only the role of a support system for decision making, but also participate in managing some processes, for this, it needs to have a component directly interacting with the management facilities. In fact, the automated dispatch control system (ADCS) is required for converting mathematical and algorithmic models into physical managing signals or commands.

Managing actions can be sent

- to programmable logic controllers for adjusting processes which they are involved in;
- to executing agents, whose role can be filled by people (workers, users, etc.) and by programs (simulators, game agents, etc.) and devices (motors, heaters, etc.);
- to other management systems (for example, of a lower level) or other knowledge-based systems.

The automated dispatch control system in general includes a management facility (or facilities), a set of devices for sending managing actions (devices for interface with object), computers (realized on the basis of programmable logic controllers, microcontroller/microprocessor systems, and other devices), a set of tools for reading the signals from the object.

Experiment component

If in a situation of uncertainty in the system there is nowhere to get data/knowledge from for further action, the only thing left is to take some experimental steps. Here we are talking about creating new experience (unlike the actions of the experience analyzer component, which works with data already existing at that moment), in other words, a trial operation with an analysis of the results obtained.

The main difficulty lies mainly in the probable danger of an experimental action, which is linked with the danger of an unidentified object, with the risk of damaging equipment due to improper action, and with a possible lack of access to the object (closed nature of the object).

Furthermore, experiments should be conducted according to a certain consistency, with certain rules for planning and conducting experiments, since the goal is to get systematic and not random results .

In connection with this, the purpose of this component is to determine the goals and conditions of the experiment, to plan it immediately, to obtain experimental data (which is then checked) to send on to be processed. The rules of conducting experiments on which this component works can be stored in knowledge bases.

It is worth separately noting that the problems of planning and conducting an experiment are directly related to problems of **generating a hypothesis** and **testing a hypothesis**. Each of these problems is rather complex and requires individual consideration. However, in the framework of this textbook, as a way to implement one of the approaches to solving the first of these problems, we will only mention the JSM-method [Gavrilova, Kudryavtsev, Muromtsev, 2016] and methods of Association rule learning. The Tableau algorithm will be looked at as an approach to solving the second problem [Zolin, 2018].

The tableau algorithm is used to prove or refute statements presented in one of the logical languages. Modifications of it exist for the languages examined here: propositional logic, first-order logic and description logic.

The approach is simple: in order to prove a statement, it is necessary to show the inconsistency of the refuting statement. In other words, if it is necessary using this approach to show $[\,(a \rightarrow b) \wedge b\,] \rightarrow a$, then we need to show that $\neg\,\{\,[\,(a \rightarrow b) \wedge b\,] \rightarrow a\,\}$ is a contradiction, an unfounded statement. Meanwhile, to prove (from description logic) that $\exists R.A \sqcup \neg\forall R.(\neg B \sqcup A) \sqsubseteq \exists R.B$, we need to prove the inconsistency of the statement $\neg\,\{\,\exists R.A \sqcup \neg\forall R.(\neg B \sqcup A) \sqsubseteq \exists R.B\,\}$.

The next steps are fundamentally the same in all three logical languages: logical statements are transformed to branches (which are called “tableaux”) and create an expanding branching structure. Next, rules (related to revealing conjunctions, disjunctions, restrictions, etc.) are applied to each of the branches. According to these rules, these branches either extend, close or remain unchanged. If at the moment the next steps of the algorithm for transforming the branches are not possible, then either all branches are closed (“blocked”), or some remain unclosed. In the first case, the statement put forward, negating the initial statement, is considered contradictory, and the initial one is proved. In the second case, the initial statement is considered to not be universal and therefore is refuted.

Of course, solving the problem of generating and testing a hypothesis is not limited to the JSM method and tableau algorithm. However, familiarity with them helps to at least sketch out approaches to solving the indicated problems.

Search component

An intelligent system should act in a situation of uncertainty, for which it can either analyze previous experience in order to derive new knowledge, or create new experience (but conducting experiments), or search for missing knowledge in external sources.

To solve the last of the indicated problems, a search component is assigned, whose functionality includes:

- obtaining information about which (about what?) data/knowledge is required;
- forming a query and sending it either to structured knowledge sources (DBPedia, Wikidata and others), or to unstructured ones (searching in literature, in a Global Network and so on);
- analyzing the search results, structuring and formalizing them into knowledge form.

Core

This is an integral, connecting element of a system.

Its main task is to organize an information link between the core and all the other components: in fact, the delivery of messages between them. Thus, the core is a set of tools for reading data/commands from their sources and sending them to their destinations.

Here it is important to understand that the core itself does not make decisions about the methods and tools for processing incoming data. An appropriate component for the task is responsible for each of the decisions. This must be taken into account when developing each of them.

Furthermore, understanding the very processes of information exchange between the components must precede the software implementation of the "insides" of the latter, as only it gives understanding about the work of the system as a single whole. For this, in turn, it is necessary to select the form of implementation of the core.

In the first case, the core is implemented as a comprehensive, independent unit, a set of functions which is independent from components. Then the components themselves only prepare data for sending, while the core, in a specific sequence, cyclically asks all components. At each iteration, one or several transactions are performed (similar to how it is done in the cycle of a programmable logic controller). Of course, here parallel computations can be carried out as well, if the hardware provides such an opportunity.

In the second case, the core is dispersed, i.e., its functions are distributed among the components. Then each of these must, in addition to its own functions, carry out an information exchange with other components. In this form of implementation, the core does not exist as an individual component.

Multi-agent organization of a knowledge-based system

Everything said above about KBS does not mean that it is a complex, completely located on a single computing device. Furthermore, if a management system is presented as a whole business, when we are talking about a large number of signals, devices, indicators, uncertainty, influencing factors, interactions, etc., then with such centralization, the performance of the whole system will largely depend on the characteristics of the central computing device and the quality of the channels joined to it, since all decisions are made centrally, which is not preferred.

This is one of the reasons for the transition to distributed systems within the trend of intellectualization, where instead of a single decision making agent device, there are several. **Multiagency** will be considered as the structure of a knowledge-based system, represented by a set of software agents on several physical devices.

An example of such a system can be the factory management system described in the work [Kovalevskiy, Onufriev, 2019], where the general functionality of a business is provided by a set of agents (carried out on the platforms of Raspberry Pi, each of which monitors its site of the enterprise), joined into one network. At the same time, each of them is a knowledge-based system, although any of them will be much simpler in functionality and design than the result of their combination, which is a multi-agent knowledge-based system.

However, potentially lifting the specified restriction, such a transition also raises questions about what form each KBS component will be appear in if there is now more than one intelligent device, and if they all implement components similar in functionality.

Possible forms: **network component, local component** and **distributed component**.

A network component is the first form of implementing a component. Such a component is not stored in most agents. It is accessed over the network. At the same time, exchanging with it is not significantly different from the case of it being "on board". An example of such a component is a knowledge base which is physically located on a single device, and which the rest of the agents refer to when needed.

The second form is a local component, the essence of which can be explained by the phrase "to each his own". This means that such a component can exist on each device, but is focused on solving, for the most part, the tasks of only this device. Accordingly, it has the content required for this device to work. An example is a database which can appear with every agent, but stores the information necessary only for it. Thus, these local databases may not overlap for different agents. However, local components may be available for the rest of the agents.

The third and most complex form is the form of distributed components, meaning that part of the last one, even if physically dispersed across different devices, nevertheless make up a single, comprehensive system. This means that it is necessary to organize interaction between the parts of a single system. So, if this is a distributed database, then its parts (which can overlap each other) should be synchronized.

The need to solve the problem of component forms is related, on the one hand, to the fact that not one agent should control the whole system entirely (the latter in this case would be centralized, making multiagency unnecessary), while, consequently, its own information may not be enough to complete certain tasks. This means that certain components should, at least, be available to other agents. On the other hand, if multiple agents simultaneously refer to the same source for information, this can lead to a high dependency of their work on the quality of the connection and the bandwidth of the channels. However, if instead of this, the components will be local or distributed (especially if the information in them is duplicate), this generates a number of problems relating to dispatching and synching data.

We will examine the components individually.

A database should not function like a network, since in this case its tasks will include collecting all real-time data from each device. At the same time, data about the work of one of the agents may be necessary for the work of another. Therefore, an acceptable form is a local DB with the possibility of other agents accessing it if necessary.

A monitoring system should be implemented in a local form and work with a local database.

The same can be said about an information measuring system.

Meanwhile, a knowledge base, depending on the potential frequency of its demand, may well be used in network form. For this component, any of these forms is possible. However, the more "local" the knowledge bases, the more complicated it will be for the agents to "understand" in which of the available knowledge bases they need to search for the values required for each specific case.

The components of management for neural networks and formulas are better to function locally, solving the problems of the device on which they are located.

An experience analyzer may not be a constantly active component, and therefore, along with a local form, a network form is perfectly acceptable. The same can be said about components of search and experiments.

The user interface can also be a network (for example, on a user terminal or in the form of a web application), as well as local – if the latter is required.

If an automatic control system is used, then it is unlikely that all devices should have access to it.

In any case, the problem about the form of implementing any component must be solved for each specific projected knowledge-based system.

It is important to keep in mind that no ready-made “recipes for producing” knowledge-based systems exist. This is still largely a scientific challenge requiring an individual and creative approach.

However, only systems solving significant uncertainties (and only them) can claim to be intelligent.

REFERENCES

1. Gavrilova T.A., Kudryavtsev D.V., Muromtsev D.I. Knowledge Engineering. Models and Methods: Textbook [Inzheneriya znaniy. Modeli i metody: Uchebnik]. – SPb.: Lan' Publishing, 2016. – 324 p.
2. Gavrilova T.A., Onufriev V.A. Conceptual Modelling: Common Students' Mistakes In Visual Representation. Proceedings of the 20th International Conference on Interactive Collaborative Learning, 2018, Vol 1(715), pp. 62-71.
3. Zolin E.E. Description logic (special course), Department of Mathematical Logic and Theory of Computation, Department of Mechanics and Mathematics, Moscow State University, http://lpcs.math.msu.su/~zolin/dl/ (20.01.2021).
4. Zyuz'kov V.M. Mathematical logic and theory of computation: textbook / V.M. Zyuz'kov. [Matematicheskaya logika i teoriya algoritmov] – Tomsk: El' Kontent, 2015. – 236 p.
5. Knigin A.N. The doctrine of categories: a textbook for students of philosophy departments. [Uchenie o kategoriyakh] – Tomsk, 2002. – 193 p.
6. Kovalevsky V.E., Onufriev V.A. Multi-agent algorithms coordinating key performance indicators of at enterprise [Mul'tiagentnye algoritmy soglasovaniya klyuchevykh pokazateley effektivnosti predpriyatiya] // Scientific and Technical Journal of SPbSTU. Informatics. Telecommunications. Management. 2019. T. 12, № 3. pp. 67–80. DOI: 10.18721/JCSTCS.12306.
7. Khokhlovskiy, V.; Oleynikov, V.; Kostenko, D.; Onufriev, V. & Potekhin, V. (2019). Modernisation of a Production Process Using Multicriteria Optimisation Logic and Augmented Reality, Proceedings of the 30th DAAAM International Symposium, pp.0500-0507, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-22-8, ISSN 1726-9679, Vienna, Austria DOI: 10.2507/30th.daaam.proceedings.067.