ORIGINAL RESEARCH

Knowledge representation with *T*

Boris Sunik*

Software Developer E3-Zuken GmbH, Ulm, Germany

Received: October 29, 2018 **Accepted:** November 18, 2018 **Online Published:** December 6, 2018
DOI: 10.5430/air.v7n2p55 **URL:** https://doi.org/10.5430/air.v7n2p55

Abstract

The universal representation language ***T*** proposed in the article is the set of linguistic items employed in the manner of a natural language with the purpose of information exchange between various communicators. The language is not confined to any particular representation domain, implementation, communicator or discourse type. Assuming there is sufficient vocabulary, each text composed in any of the human languages can be adequately translated to ***T*** in the same way as it can be translated to another human language. The semantics transmitted by ***T*** code consist of conventional knowledge regarding objects, actions, properties, states and so on.

T allows the explicit expression of kinds of information traditionally considered as inexpressible, like tacit knowledge or even non-human knowledge.

Key Words: Knowledge representation, Universal knowledge representation language, Theory of meaningful information, General information theory

1. Introduction

The discipline of knowledge representation (KR) studies the formalization of knowledge and its processing within machines.[1] It was established as a separate branch of AI during the 1970s when the first tasks related to intellectual reasoning came to the agenda. The methods of KR include production rules (if ...then ... statements), semantic networks, frames, first order logic. Of all these methods, only rule-based knowledge representation was ever used for practically relevant developments. During the eighties, it was applied in a number of large-scale projects, and found its way into many industrial enterprises and government applications.[2]

None of the other KR methods was ever used for industrial programming, also because the mainstream programming languages (C/C++, Java etc.) were essentially better in respective implementations. At this time, the development of intellectually loaded applications did not differ from programming of complex "non-intelligent" systems, as both areas require building millions of lines of code running on complex hardware systems.

Actually, we need to speak about two completely different KRs – one grounded in scientifically proven principles but failing in practical developments and another one based on the concepts of mainstream programming and demonstrating efficiency unachievable by the former, both completely distinct from each other. The only concept shared by both KRs is object-orientation.

The universal representation language ***T*** introduced in this paper is based on ideas developed in mainstream programming. Its conceptual system uses the C++[3] notational system extended by non-executable representations. ***T*** is not a programming language; it is the set of linguistic items employed in the manner of a natural language with the purpose

*Correspondence: Boris Sunik; Email: boris@generalinformationtheory.com; Address: Software Developer E3-Zuken GmbH, Ulm, Germany.

of information exchange between various communicators. The language is not confined to any particular representation domain, implementation, communicator or discourse type.

Assuming there is sufficient vocabulary, each text composed in any of the human languages can be adequately translated to ***T*** in the same way as it can be translated to another human language. Furthermore, it allows the explicit expression of kinds of information traditionally considered as inexpressible, like tacit knowledge or even non-human knowledge.

The semantics transmitted by ***T*** code consists of conventional knowledge regarding objects, actions, properties, states and so on. The language does not employ common formal models and approaches like first order calculus, semantic nets, conceptual graphs and frames traditionally used in the realm of KR. It is assumed that common knowledge notions constitute the basis of all philosophical and scientific conceptual systems allowing the representation of all these systems in the form of class hierarchies.

The conceptual basis of ***T*** is created by the Theory of Meaningful Information (TMI).[4] More on this topic can be found in Ref.[5] The view of the world defined on the basis of this theory actually enables the conceptualization of arbitrary real or imaginable things in terms of ***T*** concepts. This allows to eliminate, probably the most salient problem of contemporary AI consisting in the lack of sustainable and effective methods of knowledge representation. According to TMI, this problem results from the one-sided concentration of the AI scientific community on methods based on mathematics and logic, while completely ignoring the idea of building a commonsense knowledge representation language.

The TMI notion of knowledge used in this work doesn't adhere to Plato's understanding of knowledge, which he defined in his Dialogues as "*justified, true belief*".[6] The work follows the approach of Kant, who believed that "*the body stays as a condition of knowledge*",[7] see more in Ref.[5] Chapter 3.

This article explains the basics of ***T*** as well as the specifics of the TMI approach concerned with uniform representation of all information types. The next section contains the problem statement, followed by the general language model, a short specification of ***T***, the ontologies of event and time and the conclusions.

2. THE PROBLEM STATEMENT

2.1 The semantic gap between KR and mainstream programming

Taking into account more than 40 years of intensive research and the diversity of developed solutions, the only plausible explanation for the failure of KR science in developing truly efficient KR tools is that the proposed ways simply do not lead to an effective solution. It is not hard to find the crux of the failure.

The obvious mistake is the total disregard for the quite obvious circumstance that the effective KR can be built exclusively on the basis of a *universal knowledge representation language*. Most of the knowledge we have is represented in namely such a language. Each natural language is an ultimate KR tool, which can be used by everyone from infants learning their first words to scientists discussing the most complex theories. The only (however grave) problem is that modern computers have difficulty understanding texts composed in a natural language because these texts suffer from ambiguity and a vast amount of implicit background information.

This problem can however be solved with the help of an *unambiguous* language. Curiously, such a solution has never been seriously discussed in the scientific community. The reason is the widespread belief that the universality (expressive power) and unambiguousness are supposedly non-combinable characteristics. See, for example Sowa: "The expressive power of natural languages, which is their greatest strength, is also one of the greatest obstacles to efficient computable operations".[8]

A search for any kind of explicit proof supporting this negative stance however doesn't turn up anything except for a few banalities about the primitiveness of computers and the complexity of human knowledge in general and human languages in particular. Contrary to general belief, this opinion is not based on the particular results of any thorough language study, but from the lack thereof.

The real reason for the lack of discussion and, as a consequence, the non-development of a universal representation language is the penchant for the scientific pureness characteristic of KR science. While several decades were spent on the intensive study of scientifically proven even if practically inapplicable methods, the KR-relevant characteristics of mainstream programming languages were stubbornly ignored because the methods of these languages are supposedly "unscientific".

Since the unscientific methods are not discussed in the scientific literature, no references can be provided. However, it is exactly this characteristic of the mainstream programming languages that I receive as a reply every time when questioning specialists regarding why they don't study the KR features of these languages. It is clear that formal languages with such expressive power and effectiveness cannot be characterized in this way, unless it is an excuse for suppressing undesirable discussions.

In fact, such a discussion is merely impossible, because the conceptual world of KR simply lacks the necessary concepts. The KR science was created and developed as a branch of a theoretical computer science. Its means of choice are methods and models developed in the framework of the latter, but the notion "programming language" used by computer science has nothing to do with programming languages actually used. The branch of computer science responsible for the content of these notions and known as "the theory of programming language" never studied the programming languages actually used, even though the name of this theory suggests otherwise. Its subject was the mathematical and logical basis of programming, which is something that does not exist in reality.

Real programming is a pure construction activity that has nothing in common with mathematics. A programmer creates a program using FORTRAN, C++, PHP or any other programming language in the same way a bricklayer builds a house using a trowel, bricks and mortar. Surely there are cases when mathematical knowledge is necessary like operations with graphics whose transformations are represented by mathematical means. Generally-speaking, a programmer does not need any more mathematics than a good ninth grade student.

The fact that theoreticians of programming see it differently is very well demonstrated by the article "Programming Language Theory" in Wikipedia. The article enumerates 27 of the most important works in the area, none of which is devoted to mainstream programming languages which are merely mentioned as a rather outsider phenomenon. Of the four people who have made the biggest contributions to programming: John Backus – the developer of Fortran and BNF, Nicklaus Wirth – the creator of Pascal, Dennis Ritchie - the creator of C and Bjarne Stroustrup - the creator of C++, only Backus is mentioned. The reason is a class of function-level programming languages[9] which he proposed in 1977. This work was highly valued by the scientific community but produced zero impact in actual programming.

The reasons for this development are rooted in early computer history. The first computers were created during the Second World War with the purpose of mathematical calculations for military needs. This development was intensified after the start of the Cold War, so it is not astounding that many programmers of that time were mathematicians and logicians who viewed programming as a branch of mathematics.

Their theoretical preferences, however, played no role in the first decade of programming. Direct programming in machine code was a very exhausting and slow process, so the programmers had to ponder about alternatives, which consisted in the development of programming languages. The turning point was the development of the first high level programming language FORTRAN (Formula Translator) in 1956, which for the first time allowed programs to be written oriented on its logic and not on the implementation of machine code. Other high-level programming languages followed soon after and the creation of the theory came to the agenda.

During the following years, the practitioners became theoreticians and produced a rigorous mathematized theory, which basically ignored all non-mathematical approaches supposedly as non-scientific. Often the same person contributed to both areas, as did John Backus who designed BNF and led the team that created FORTRAN – two decisive developments, which created the modern programming we know today. He also defined a new class of functional programming languages, which was completely in accordance with all the rigorous requirements of programming theory, but useless from a practical viewpoint.

The fact that the theory does not cover the characteristics of the practically usable programming languages was then seen as a negligible problem, because these languages were viewed as a temporary solution needed for bridging the time until the maturing of the theoretically correct programming tools. Neglecting mainstream programming went so far that the theoreticians failed to even produce a generally accepted definition of a programming language because such a definition could not be made in the conceptual coordinates of recognized theories.

The lack of the adequate definition of a programming language was thoroughly described by Jean Sammet as early as 1968 in her famous book "Programming Languages: History and Fundamentals".[10] The described status quo has not changed since that time. Most modern books about programming languages simply avoid programming language definitions directly appealing to the implicit meaning of these notions e.g.[11] Others, obviously guided by reasons of scientific politeness, again produce non-informative definitions like "any notation of algorithms and data-structures".[12]

A science unable to provide an adequate definition of its own subject hardly has any chance of surviving and programming language theory was no exception. Its ultimate failure coincided with the end of the Cold War.

Scientific research, intensively supported in times of scarce computer resources and sumptuous financing of the Cold War, lost any appeal after the latter's end. The state support was finished and most of the previously unsolvable software problems were solved with the help of Moore's Law or were on the way to that in the foreseeable future. As a result, the programming community quietly distanced itself from the obsolete theories, completely concentrating on pragmatic solutions.

While the lack of a relevant theory did not hinder the development of mainstream programming languages, it disallowed their understanding. The high knowledge potentially behind these complex systems degraded into acquired skills like driving a car. Now, the only way in which a programming language can be acquired is learning by doing. While admitting that this is a highly effective method for learning skills, the concepts formed in the heads of specialists in this inductive way are based on tacit knowledge and do not allow deep scientific conceptualization.

2.2 Voiding the semantic gap - The adequate definition of a programming language

The ineptness of the programming language theory in defining its subject however does not mean that such a definition was never created. Actually, it was! Moreover, it is one of the oldest definitions of a programming language ever. However, it was never officially recognized by computer science because of its non-mathematical background.

Here is the oldest formulation of an adequate definition to be found in the ACM library: This definition was made by Saul Gorn in the short article in the year 1959,[13] "The point of view expressed in this paper makes more tangible two principles accepted intuitively by many programmers and logical designers. They are: a) the equivalence of formal languages and machines, b) the equivalence of programming and hardware."

The same idea was detailed in 1978 by Yaohan Chu[14] "To each programming language, there is associated an ideal computer architecture which executes the program written in this language. This ideal architecture images the control constructs and the data primitives of the programming language. It is a virtual architecture, because it may not be possible to be fully implemented by real hardware architecture. If the programming language is a high-level, the virtual architecture is a high-level architecture."

Because I was not aware of the definitions above at the time of writing this paper,[15] I used my own definition, which is basically the same: "A programming language is essentially the machine code of the processor implicitly introduced by the definition of this language."

The common point of these definitions is the interpretation of a programming language as a combination of two things differing by their nature: a **genuine language**, which is a system of linguistic signs used in some communication and a **processing unit** executing programs composed in this language.

A processing unit is a pure logical construct that is not purposed for whatever physical implementation. Instead, a source program is either directly executed by a software interpreter running on a physical low-level processor, or is translated to the machine code of a physical processor with the help of a compiler.

A genuine language is a linguistic tool of limited usage. Its purpose consists of instructing a processor and its representation world is restricted to the depicting bit sequences allocated in the computer memory and the sequentially organized manipulations with these bits.

In this paper, a logical processor implicitly given in the language definition is designated as an innate processor and a computer powered by this processor as an *innate computer*. The functionality of an innate processor can be completely emulated by a plain non-optimized language interpreter running on the top of a real low-level processor. An innate computer can be simulated with a real computer executing source programs with the intermediary of such an interpreter.

For the purpose of this article, the difference between a real and an innate computer can be reduced to the differences in the executable files. Assuming that both computers have an identical application `HelloWord`, in order to run this program on a real computer, a user has to compile the file `HelloWorld.cpp` into the object file `HelloWorld.obj`, link the latter getting the executable file `HelloWord.exe` and start running the executable. Because the processor of an innate computer understands C++ code, a user of such a computer simply starts `HelloWorld.cpp` without any additional preparation.

3. The language theory

The complete theory of language can be found in TMI[5] Chapter 6. The theory describes the general language paradigm, which is the pattern of language definition and use. The paradigm represents the real life of a language — the way it is created, used, extended and substituted. The paradigm also shows the general way to formal representation of semantics, which consists of formal representation of communicators.

3.1 The definition of language

The approach of TMI is based on the commonsense understanding of language, which, in slightly different forms, is replicated by every dictionary. For example, Britannica defines language as "a system of conventional spoken or written symbols by means of which human beings, as members of a social group and participants in its culture, communicate" (the article "Language", in: Ref.[16]). The categories normally associated with a language are primarily those involving the process of language study and use, as syntax, semantics, vocabulary, the level of language control and so on.

This work uses the generalized notion of the above definition, which is not restricted to whatever particular language type and can be applied to any computer, human or even non-human language. A ***Language*** is a system of admissible values of some mediating object (variable) used for coordination between communicating entities. Communicating entities can be human beings, animals, computer algorithms and any other entity with the ability to produce and assess the value of a mediating object. A value of a mediating variable is a communicated message.

3.2 The basic model

The following examples in C/C++ demonstrate features of the simplest languages detailing the understanding of language used in this work.

1) The following simplest imperative language allows only two commands, which are the values 1 and 2 of the variable `action`. A part of algorithm, which sets the value of `action` communicates with the part of algorithm presented with a `switch` statement. Command 1 causes the execution of the procedure `a1()`, command 2 invokes the procedure `a2()`.

```
int action = 0;
        ...
action = 1;
        ...
switch (action)
{
 case 1:
        a1();
        break;
 case 2:
        a2();
        break;
 default:
}
```

2) The following language extends the previous example by separating communicating algorithms. The only difference from the example above is the way of commanding, which involves the function `doAction` actually invoking a switch statement. The language consists of a value of the parameter `action`, while the function name "`doAction`" belongs to the outside context of the communication environment.

//file1.cpp the IDE part

```
void doAction(int action)
{
        switch (action)
        {
          case 1:
            a1();
            break;
          case 2:
            a2();
            break;
          default:
        }
}
```

//file2.cpp, the ISE part

```
extern void doAction(int action);
doAction(1); //the command 1 is executed
```

The specific of this example is information transmission with the help of two consecutively processed mediating variables: the actual parameter with which `doAction` is invoked and its copy on the call stack actually used in the function body under the name `action`.

Sets of mutually convertible values are fundamental in any communication. Thus, a C++ letter 'a' has multiple alternative embodiments actually involved in the communication process:

- a written letter, which is a picture drawn on some hard surface (paper);
- a representation of this picture in the programmer's brain;
- a binary structure representing this picture in the binary computer storage (a `char` literal 'a');
- a structure of pixel states projecting the same picture on the display screen;
- a representation in the external data carrier like a hard disk of a flash memory; every data carrier can have its own coding system;

3) The following example demonstrates the simplest non-executable language. The language, represented by the type `bool`, has only the two sentences `true` and `false`. The variable `bVar` is an information carrier containing a text composed in `bool`.

```
int   num1, num2;
bool isEqual(int , int);
...
bool bVar = isEqual(num1, num2)
...
if(bVar == true) foo();
```

The output value produced by the function `isEqual` designates a meaning.

The sentence "`true`" means "*num1 and num2 are equal*", the sentence "`false`" means "*num1 and num2 are not equal*".

A text in `bVar` is a copy of the original value produced by `isEqual`. While the original sentence is an assertion proving the fact of equality, a copy is an assumption prone to possible distortions, because it can also receive a value as a result of a simple assignment "`bVar=true`" without any invocations of `isEqual`.

A message in `bVar` contains indicative information, which is assessed by the algorithm represented by the sentence `if`.

Meaning is a value of some type produced by a comparison procedure. A comparison procedure is the simplest observer. It compares an arbitrary number of entities, returning the designation of the comparison. The complex meaning is the collection of simple and complex meanings. Complex meaning requires complex observers including many comparison procedures.

The semantics associated with a language code includes several components; Thus the complete semantics of the value '1' in the variable `action` (examples 1, 2) include the semantics of the referred procedure `a1()` as well as the semantics of the code containing the `switch` sentence.

The semantics of the text in `bVar` (example 3) includes the semantics of the code writer (`isEqual`) but can also include the semantics of the complete algorithm, if `isEqual` is logically connected to other code parts. Additionally, it can include the semantics of a code reader in case the comparison procedure `isEqual` used by the latter is different from `isEqual` from the code writer.

The differences between indicative and imperative semantics can be seen in the statement "Pete puts the cup on the table". Below is the meaning of the indicative variant:

- The sentence describes putting a particular cup on a particular table.
- The process can occur everywhere, anytime or even be imaginable.
- The process is not confined to the producer or a reader of this sentence.
- It is implied that anyone reading this sentence has to understand its meaning (semantics).

The semantics of the imperative sentence "Pete, put the cup on the table" is as follows:

- The sentence is a command, which its producer gives to its reader Pete.
- The designated process can only occur if its reader is able to take the required cup and put it on the required table.
- The designated process will only occur when its reader actually executes the command.
- A code reader may also execute the command without understanding the semantics of the sentence, e.g., when the code reader is a robot and this command is one it can carry out.

To summarize, a reader of an indicative code has to be able to understand the meaning of the message, but does not need to have a physical connection to the action's environment. On the other hand, a code reader of an imperative command may act without understanding the command's meaning, but is able to actually produce the required actions.

3.3 A universal representation language

The reason why creators of programming languages never showed any interest in their KR capabilities, lays in the self-sustainability of programming whose informational structure has remained the same since the time of first computers. The main participant in the programming process is a programmer who gains knowledge with the help of natural languages, in the form of various images or as tacit knowledge. A programmer uses the knowledge acquired for issuing unambiguous instructions to an innate processor.

In the circumstances where all complex knowledge-related activities are effectively performed by humans there is absolutely no stimulus for extending a programming language to the tasks laying outside of its original purpose. One of consequences of this approach is the paradoxical situation in which a language used for producing formal representations can only be described with the help of informal categories. Concretely, while the meaning of the character literal '`a`' is unambiguously defined in the language's definition, the meaning of the letter '`a`' in the language alphabet is completely undefined and hence misunderstood.

The adequate definition of a programming language solves this paradox by revealing the missing links between the real world and the binary entities represented by the language code.

The communication in a programming language occurs between a programmer and an innate computer. A programmer studies a language by reading the definition of this language (let us assume the definition is given in a readable format) and applies it by composing a program, which he sends to an innate computer, in turn executing the commands. In OO terms Figure 1, this communication environment includes an object Programmer generating a Program; an object PLComputer, executing the Program; an object Book, from which a Programmer studies the language; a class Language, which is the multilevel structure based on alphabetical characters used in the communication exchange between a Programmer and a PLComputer.

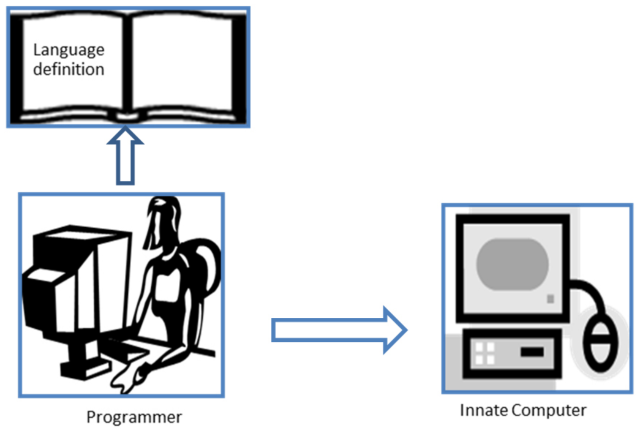

Figure 1. The communication environment

Even though the system of programming language signs is dedicated to the representation of a tiny fragment of the real world, similar to any other jargon it is unrestrictedly extensible. Taking into account that all currently existing full-fledged natural languages were built from the primitive jargons used by ancient societies, *every programming language can be considered as the embryo of the respective universal representation language.*

A system of signs of a programming language is dedicated to the representation of a tiny fragment of the real world. So it is essentially equivalent to a jargon, which is a subset of a natural language restricted to particular communicators and a topic. The fundamental characteristic of jargons is their unrestricted extensibility. Taking into account that all currently existing full-fledged natural languages were built from the primitive jargons used by ancient societies, *every programming language can be considered as the embryo of the respective universal representation language.*

The latter is not restricted by whatever predefined communicators or communicated themes and can be used for expressing any information ever formulated in any human or computer language. Such a language can be built by extending the linguistic signs of a programing language by expression abilities needed to represent entities exceeding the representation world of this programming language.

Theoretically, the world of universal representation languages can include many of them, but in reality, such plurality is rather disadvantageous, because all these languages will ultimately possess exactly the same expression power. The programming languages are very different in their expansive abilities. Some programming languages are more extensible than others with the best match being the maximally extensible of them - C++.

4. The universal representation language *T*

4.1 General characteristics

T is a formal system of C++-like signs purposed for information exchange between various communicators. The language allows the representation of both executable (imperative) and non-executable (indicative) semantics. Its default code form is indicative.

The communicators are either physical objects (human being, devices, computer programs) or logical entities, which are (or considered to be) able to read and/or write code composed in ***T*** and execute designated actions. A human being is the ultimate communicator possessing all the functionality of the language, but other communicators could possess limited abilities.

The language exploits the TMI view of the world, according to which, every real or imaginary entity can be viewed as an object whose behavior can be expressed in the manner of programming algorithms. For example, the algorithm of the Earth consists of the cyclical movement around the Sun and the rotation around its axis each day. In the same way the structure of the Sun, Earth and their algorithms can be described in English they can also be described in T.

The expression means of ***T*** contains practically all C++ concepts including real, abstract and virtual classes, instances, procedures, multiple inheritance, templates, control statements (called sentences) etc. It has also several new concepts, which are necessary for non-executable representations.

The language formalizes all parts of speech, nouns are represented as classes and instances, verbs are procedures, rules for other parts of speech are described in the following subsections.

Because of new concepts and irregularities of C++ syntax, the syntax of ***T*** is only somewhat similar to that of C++.

T allows multiple implementations. An implementation of ***T*** is its application in some communication environment where the whole language or some subset of it is used for the information exchange.

The ability of a language to specify its own grammar and semantics (meta-representation) is essentially similar to that of natural languages (English, Russian, Chinese etc.). The language is its own meta-language using the same expression means for both the specification of a communication environment and for composing discourses exchanged in this environment.

The method for meta-description is detailed in Ref.[17] The formality in ***T*** is understood as the compliance between the formal and actual parameters of the respective procedures. The check for the compliance between the formal and actual parameters is the only action generally defined for the processing of ***T*** code. This is completely sufficient providing all entities referred with the help of this language are described in the terms of OO categories.

The following specification is restricted to the description of features used in this article. The complete specification can be found in Ref.[5] Chapter 6.

4.2 Expression power

The expression power of ***T*** can be shown using the following C++ example:

```
int i1 = 5.0;
Complex c1 (3.0, 4.0);  //a complex number
consisting of two real numbers
```

The C++ meaning of this code sums up the semantics originating from four different sources.

(1) Language documentation. The object `i1` is an instance of a fundamental type `int` whose description can be found in the definition of C++, composed in a natural language, normally English.
(2) C++ code. The object `c1` is an instance of a user-defined type `Complex` whose complete definition is made in C++.
(3) Foreign knowledge. While the names `int` and Complex designate binary objects, their semantics are essentially extended by the associations with the mathematical theory defining integer and complex numbers. Similar to the C++ documentation, the description of the mathematical theory is also made in a natural language.
(4) Comments. The text after "//" gives additional information in English.

The code above can also be interpreted as a ***T*** code completely emulating C++. Such a reinterpretation is semantically indifferent but it radically changes the associated lexical environment, because all notions, which in the case of C++ are made in a natural language, can now be made in ***T***. In contrast to C++, ***T*** allows all aforementioned specifications including the complete specification of the C++ innate processor, the mathematical content and comments.

In actual fact T allows creation of a monolingual communication environment in which it could be used for expressing miscellaneous programs and narratives.

4.3 The semantics

The code semantics are basically defined by the features of communicators. The semantics of imperative code are those of a code reader (execution unit). The indicative semantics represent the features of an observer, who is a coder writer.

The ultimate observer is a human being. It collects information with the help of its senses and exchanges this information with other human beings in a human language. The model describing the information capabilities of a human as Homo Informaticus (HI) can be found in Ref.[4] The main points of this model are as follows:

- The model of an HI perceives a human being as a self-programming system commanded by the controlling computer (brain). The body of an HI is seen as the complex set of moving parts, each consisting of a muscle attached to a bone or an organ. A muscle executes one of two commands contract or relax, which, in turn, causes a movement of a bone or organ. Another important part of this system is the senses, which consist of multiple receptors interpreted as volatile variables functionally similar to the input ports of conventional computers.
- Human knowledge is represented by the programming code and data (perceptions) of an HI. The model provides the method for representing complete human knowledge starting from low-level perceptions that humans acquire during their lives and continuing until encompassing high-level knowledge. Various kinds of knowledge are interpreted as programs of distinct levels. The built-in algorithms of the lowest level are unconditional reflexes and basic skills, the low-level soft programs designate skills, and the algorithms of the high-level stand for beliefs, concepts, etc.
- The intellectual operations performed by the HI's brain are interpreted as manipulations with the code of these programs, and a natural language is considered as the system enabling programs' exchange between various individuals. Only programs of a high level can be

exchanged in a natural language. Other programs constitute tacit knowledge, which can only be transmitted with the help of immediate learning if at all when one acquires new skills by copying the behavior of another HI.

4.4 Communications

An application can include many modules. A module is a text composed in ***T***, which includes a sequence of declarations, definitions, labels and commands devoted to a particular subject. A module can be a paper book, a book's part; a file in the computer storage, a file's part.

A module can include narrative and executive information. Narrative information (*info*) is essentially a value or a collection of values describing something. Executive information (*rule*) requires execution of actions of some form from a module's reader.

Communicators can be referred to in ***T*** code with the help of predefined names `reader` and `writer`. Events of reading and writing are designated as respectively `read`, `write`. The features of communicators can be completely described in code and such description, if present, constitute the basis of code semantics. The differences between C++ and ***T*** can be shown using the famous "Hello World" example:

```
void main()
{
   printf("Hello␣World");
}
```

A ***T***-application with the same semantics includes the same code extended by the specification of communication environment. In this case only a reader of this text has to be provided:

```
_reader := _cplusplus // the definition
of the code reader
proc main()
{
   printf("Hello␣World");
}
```

Similar code can be used in completely different communications. Assume there is a logical processor called `cplusplus_logger`, which registers operations performed by an innate processor of C++ and writes a log file. A log produced during execution of the aforementioned example can be grammatically similar to the latter but it is a narrative enumeration of already executed functions.

```
_writer := cplusplus_logger
main()
{
   printf("Hello␣World");
}
```

The differences between imperative and indicative code can be demonstrated in the example of the procedure `foo` defined as

```
proc foo()
{
   foo1();
   foo2();
}
```

In ***T***, the text "`foo()`" means not the invocation of `foo` as it normally is in the case of programming languages, but the declaration of an occurrence (event) of `foo`. An event is an instance of a procedure and can also be declared in the same way as an instance of a class. The following code shows a sequence of two `foo` events.

```
{
  foo ff(); //an event of foo named ff
  foo();    //an unnamed event of foo
}
```

If an occurrence of `foo1()` in `foo()` was named as f1 it could be referenced as `ff.f1`.

The command requiring execution of foo has to be written as.

```
\#foo();
// \# designates a command, T has no
preprocessor in the C style
```

A command's executer will read the command body, interpret each procedure occurrence as a command and execute it. A name of an occurrence is interpreted as a command's name.

The object `cplusplus` executes code in exactly this way. The procedure, which invokes a C++ innate processor, finds the file HelloWorld.cpp and invokes an instance of `cplusplus` with this file as a parameter. An instance of `cplusplus` looks for the procedure `main()` and executes the built-in command #`main()`, which starts reading the function's body and executing its content.

The details of non-executable code are given in the following sections.

4.5 *T* Specifics

The type system of T utilizes the classical C++ concepts as classes, structs, unions and enums, employing several new constructs. The complete list of concepts can be found in Ref.[17] The extensions of the traditional C++ concepts described below are relevant for the current article.

1) ***T*** uses the concept of a heap, which is an unorganized plurality of entities. The following is the declaration of the heap hp of `int` binary numbers

```
int.. hp;     //
heap hp int; //alternative declaration
```

Heaps are the simplest pluralities. Thus the phrase "the flowers blossomed" describes a heap of flowers that blossom. Heaps allow a specification of relations between pluralities. The phrase "several village inhabitants" designates a small sub-plurality of the complete village population. See Subsection 4.7 for examples.

2) The concept of `enum` is a generalized, relatively respective C/C++ construct. In ***T*** `enum` is a restriction of the base type to a list of admissible values. Thus, the C++ enum

```
enum currency {dollar, frank, euro};
```

is interpreted in ***T*** as a restriction of the type `int`, which is named `currency` whose only admissible values are values `0,1,2` named respectively as `dollar`, `jen` and `euro`. In ***T*** this enum has to be represented as

```
enum currency int{ dollar, franc, euro};
```

A definition written in the way of the above C++ enum as

```
enum currency {dollar, franc, euro};
```

defines the set of regular words, which could be associated with a particular meaning somewhere after the definition point. See examples in the following sections.

3) Universal Declaration Syntax

All objects including procedures and arrays have explicit types. Thus, the "C" function declared as "`intfoo(inti)`" in ***T*** has to be expressed as

```
cfunc foo(int i,out int);
```

The type `cfunc` is the class of C++ functions. A declaration of the "C" array e.g., "`intii[3]`" can be made as "`carrayiiint[3]`" where `carray` is the class of "C" arrays.

4) The dot operator "." allows referencing parts of arbitrary containers. Thus, the expression 'a.b' is a correct reference for the following components:

- Procedure parameter: `proca(intb);`
- Template parameter: `proca<typenameb>(bc);` //a template parameter b is just additional parameter
- Class component: `classa{classb;};`
- Namespace entity: `namespacea{intb;};`
- Component of complex enumerations. See Ref.[5] Chapter 7 for examples.

4.6 Information statements

The basis construct used for representation of narratives is a literal structure called **information statement** (abbr. info statement). An info statement represents the complete information associated with the execution of a comparison routine. Its syntax is similar to a function call extended by the result of a comparison as its first parameter. Thus the function `isEqual` allows two info statements

```
int  num1, num2;
proc isEqual(out bool, int, int);

isEqual(true,  num1, num2); // this info
sentence means num1 is equal to num2
isEqual(false, num1, num2);  // this info
sentence means num1 is not equal to num2
```

If the comparison routine produces a unique value, the procedure name can be replaced by the name of this value.

```
enum equility int{equal, nequal};
proc isEqual(out equility e, int, int);
nequal(num1, num2);//this indicative sentence
means that "num1 is not equal to num2"
```

4.7 Representation of parts of speech

4.7.1 *Adjectives, adverbs*

These parts of speech are represented by a special form of a comparison routine returning a lexical word. Its syntax is

```
info <returned enum>(<param1> [, param2,
...paramn])
```

This kind of info statement can be written in the function-like notation for every number of parameters, as a binary operator in case of two parameters and as a unary operator for a single parameter. The function `isEqual()` can be redefined as follows:

```
info {equal,nequal}(int,int);
//equal, unequal are not names of int but
legal T words
num1 equal num2;
//the indicative sentence means "num1 is
equal to num2"  binary operator
num1 nequal num2;
//the indicative sentence means  "num1 is
not equal to num2" binary operator
equal( num1, num2);
```

```
//the indicative sentence means "num1 is
equal to num2" functional notation
nequal( num1, num2);
//the indicative sentence means "num1 is
not equal to num2" functional notation
```

The statement with one input parameter can be written in a function-like notation or a unary operator in a postfix or prefix form. The default form is prefix. The following info routine has to be produced during the comparison between the actual grass color and the predefined set of available colors.

```
enum Color (black, white, green, brown,
yellow, red, orange};
info Color(Object);
into Color(out, Object); //the same as before
green grass;
green(grass);
```

An unary postfix operator needs to be declared as:

```
info Color(Object, out); //second variant
grass green;
```

All adjectives and adverbs can be defined in this way.

4.7.2 *Quantifiers and prepositions*

Info statements "all", "some" are essentially relations between a heap and its subset:

```
info {all, many, some, few, undef}
<typename T>(heap<T> subset, heap<T>
plurality);
Student.. campusStudents;
Student.. &distinguishedStudens;
Student.. &commonStudens;

distinguishedStudents few campusStudents;
commonSstudents many campusStudents;
```

The info statement "from" characterizes the relationship between an entity and a container

```
info {from, none} (Object part, Object obj);
struct S{int i1,i2;  float f;}
//the container S includes i1, i2 and f
float from S;      //designates f from S
int.. from S;      //designates i1 and i2
objects from S

Object my_university; //includes buildings
and persons unnamed heap including
all students from my_university
Student.. from my_university;
// unnamed heap including some students from
my_university
Student.. some Student.. from my_university;
// heap with a name stdnts, includes some
students from my_university
$Student..$stdnts some Student..from
my_university;
// the same as before but without explicit
designation of a type
$$stdnts some Student..from my_university;
```

The info statement "on". Assume there is a wall with some picture hanging on it. So we can make several statements relating to the things placed in vicinity of this picture.

```
Surface wall;
Picture pic;
info {on, unrelated} (Object obj, Surface
surf); //obj is on this surface either
attached or fastened pic on wall;
```

The statements regarding parts of the wall "above" and "below" a picture can be made in the following form.

```
info {above,below}{Object part, Object
relpoint, Object complete);
Surface above(pic,wall);
//the fragment of wall above pic
```

The info statements "near", "far".

```
info {near,far}{Object ob1, Object relpoint);
Tree tree;
Gate gate;
tree near gate;
```

The following info statement describes time relationships between events

```
info {during, before, after}{event ev1,
event ev2);
event IamEating;
event TVNews;

IamEating during TVNews;
```

Other relations between events including grammar tenses are described in Section 5.1.

4.7.3 *Verbs*

Verbs are events (occurrences) of procedures. The basic construct used for expression of procedures is an application.

An application is a block statement qualified by some entity, components of which are visible in this block. The entities defined inside the block can be referred with an applied object followed by the double colon "::". The qualifying object can be either type or instance. An application is not a member of the applied type.

```
class Obj {int a; };
Object obj;
Object::{a = 0;};
// an application of an Object
proc Object::init(){a = 0;};
//init is an application of the class Object
proc obj::init(int pa)(a = pa;};
//init is an application of the instance obj

obj::init();  //occurrence of Object::init()
obj::init(7); //occurrence of obj::init()
```

The elementary events are represented in the form of changes, which are alterations of a result produced by the same comparison routine. Thus, the phrase "*the apple grows*" assures that the size of the apple in the moment of producing this phrase is bigger than before. The following code represents the growth of an apple during some time. Let's assume that the size is given in millimeters.

```
class Apple
{
   int size;
   ...
}
Apple apple;
Apple::{size = 20; size = 25;}
//assume the size is given in mm.
```

In ***T***, this change can be represented as follows:

```
proc apple::grow(){size(20);size(25);
//the declaration
apple::grow();
//an occurrence of this change
```

Simple events are essentially series of changes. Thus, a clenched fist is a series of changes of this fist, a turn of a fan is a series of changes of a fan, the working of a motor is a series of changes in a motor.

Verbs can be used in active and passive forms Thus, the sentence "*Peter puts the cup on the table*" can be expressed in ***T*** as follows:

```
Pete::put(table, cup);
```

The passive form of this sentence "the cup is put on the table by Pete" has the following syntax:

```
cup, Pete::put(table,\_noun);
//\_noun is the predefined reference to
the sentence's noun
```

The syntax of a sentence can be found in Ref.[5]

4.7.4 *Pronouns (References)*

Pronouns are represented as references (temporary names) used to refer to an object. A name is an identifier permanently assigned to an entity while a reference is its temporary identifier. C++ allows definition of references as:

```
int num = 0;     //num is an instance of int
int &rnum = num;//rnum is a reference to num
```

Differing from C++, ***T*** allows multiple assignments to the same reference. An assignment occurs with the help of the operator ":=" or "()". In the code below, the reference `bref` first designates `first` and then `second`

```
int first;
int second;

int &bref := first;
bref := second;
bref(second);  //the same as before
```

The reference assignment also has a mirrored variant "=:" which references the left side object.

```
second =: bref;  //the same as before
```

When a reference is initialized in the declaration point, it can be defined implicitly without the character "&". References defined in this way cannot be reassigned again. The following definition introduces the reference which cannot be changed

```
int constRref := second;
constRfef     := first;  //Wrong
```

Parameters are references by default. A number of predefined references have special meanings. The references `I` and `you`, are set implicitly. They designate the text creator and the current text reader. References `_write`, `_read` designate events of writing and reading the current text. A reference `that(type)` designates the last event/instance of `type`. For example:

```
class Man    :Human;
class Woman  :Human;
Man Alex;
Woman Ann;
Man Peter;

that(Woman)        //designates Ann;
that(Man)          //designates Peter;
that(Human)        //designates Peter;

Ann::work();
Alex::work();
```

```
that(Human::work);     //designates Alex;
that(Man)              //designates Alex;
```

Such nouns as n*eighbor, patient, chief* etc. are also references to real objects.

4.8 Associations

The common way of representing entities like a person consists of defining the class "Person" whose components are a person's attributes like age, position, clothing size etc. While this is very effective way of defining distinct entities, it is incorrect from the viewpoint of actually existing relations between persons and their attributes. The only parts of a real person are the person's body parts and internal organs while all aforementioned attributes are external objects related to this person in one way or another. ***T*** provides the concept of association that allows referring to entities by their features and affiliations as "his mood", "the smoking pipe of Hemmingway", "a house in London". Associations are the most fundamental kind of relationships. All other relationships like relations between an object and its parts, between a class and its method are based on this concept.

Associations are defined as relations between entities defined on the same level. The objects `name` and `shirtSize` are associated entities:

```
string name;
char shirtSize;
```

The relations between these entities can be expressed using the following two constructs:

- The horizontal qualifications between associated elements refer to the object `shirtSize` as `name@shirtSize`.
- The association constraint is used for expressing dependencies existing between association members. Thus, the meaning of the phrase "*Peter's shirt size is 'M'*" can be expressed as name("`Peter`")@[`shirtSize`(M")];

In most cases, associations are used for expressing dependencies between objects participating in the same algorithm. Consider the following example:

```
class Person {...}
class Cloth  {...};
proc measureClothSize(Person &person,
Cloth &cloth, out char size);
...
Person Peter;
```

There are several ways to express it. One is to define a respective info statement.

```
measureClothSize(Peter, Shirt, M);
```

Another way consists of redefining `measureClothSize` as an unnamed procedure.

```
proc (Person &person, Cloth &measuredCloth,
out char size); //the unnamed procedure is
defined as a synonym of measureClothSize

Peter@[cloth(Shirt),size(M)];
// first specification variant
Peter@cloth(Shirt)@size(M);
// second specification variant
```

Associations can be defined in the form of attributes, which reference the associated object only through the head entity. The following is the association between `name` and `shirtSize` expressed in the form of attributes.

```
string name [char shirtSize;];
name;                //OK
name@shirtSize;      //OK
shirtSize@name;      //Wrong. shirtSize is not
defined as a sovereign entity.
```

The number of attributes is unlimited and many of them can be referred to in the same sentence.

```
string name[char shirtSize; Date birthDay];
name(Robert)@[shirtSize(M);birthDay
(1.1.1980);]
```

5. Example ontologies

The presumption of this work is that the ontologies of event and time defined on the base of other approaches (Refs.[18–22] etc.) are expressible in terms of the commonsense ontologies below.

5.1 Events

An event is an occurrence of a procedure. It consists of a change or a sequence of changes and inclusive events.

```
proc prc();
//declaration of the procedure prc
prc();
//unnamed event of prc
```

A change consisting of similar states represents a process of perpetuation of something, e.g., "a cat on a mat" means the place of the cat is not changed during observation time.

Representation of events essentially depends on the code form. While the event implementation (algorithm) is important in the imperative code, it is normally not true for the narrative representation. The general way of narratives consists of expressing relations between specified and related events. For example, the phrase "he came home at the end of the day" denotes a coincidental time between the process of someone coming home and the day ending occurring at this moment. The event (the day ending) is considered as being the longer of both processes. So the exact semantics of this phrase is that an event of coming home occurred somewhere *during* the ending of the day.

The phrase "I was at home" implies a connection in time of the event of me being at home relative to the event of producing this phrase.

The phrase "the train leaves at 15.00" means that the train departure occurs simultaneously with the clocks showing 15.00.

The phrase "Bill drove 15 hours" in turn means that the duration of Bill's drive lasted the same amount of time as a clock needs to count 15 hours.

The declaration below introduces essential attributes of an event, which relate to the designated event in this or another way. The type `event` is a basic type, its definition is impossible without a recursion, so it is essentially a pseudo-declaration. The predefined reference `it` is used in the brackets for referring to the master object. The reference `it.it` allows referencing the master object of the encircling level and so on. The master object is the default first parameter.

```
proc event
[
   //Associated events start and finish are
parts of the main event.
   event &start; //start of the main event
   event &finish;//finish of the main event
   event &occur; //synonym of the main event
   //An event ev is an independent event
occurring simultaneously with the main event
   info {during, before, after}{event ev2);
   info (was,is,will}{);
   info (at,false}{event ev1);
   info {lasts ,false}(event ev1);};
   info {before,during,after}(event ev);
   //event enclosing includes the main event
   info {in,false]{event enclosing)
   [
          //the following code define
relations between respective events
          (enclosing@start before it.it@start
|| enclosing@start during
          it.it@start)
          &&
          (enclosing@finish after
          it.it@finish || enclosing@finish
          during it.it@finish);
   ]
          //following procedure compares the
main event this with the time of the text
production
          Place place[it.it on it;];
          //the place of events occurrence
         ]
```

Grammar tenses establish a relationship between the time of producing the sentence by a code writer and a time of an event. The assertion registering the event's tense can be defined as:

```
info (was,is,will}{event ev)
{
           extern event _write;
           if( _write during  ev )
               return be;
           elsif( _write after ev )
               return was;
           else return will;
}
Bob::goto(cinema),_v was;
// means Bob went to a cinema
Bob::walk@[be]();
// means Bob walks
```

5.2 Time

A time (date-time) is a state of a clock. A clock is a device implementing four general procedures: `count`, `stay`, `reset`, `stop`. The procedure `count` refers to the counting procedures `Second,Minute,Hour,Day,Month,Year,Era`. The value stored in `seconds,minutes,hours,days,months,years` represents the current clock's time.

A state of a clock is an event of persisting a current state. The event finishes with the change of the smallest component (e.g., the state 1.1.2008 13:30:31 ends when a number of seconds changes to 32).

```
        class Time
        {
        public:
            //type integer is the language number consisting of digits
            integer years,
                months ,
                days ,
                hours ,
                minutes ,
                seconds ;
        time(){ years = months = days = hours = minutes = seconds = 0;}
        time(integer seconds = 0, integer minutes = 0,integer hours= 0,
        integer days= 0, integer months = 0,integer years = 0);
    }
    class Clock;
    //the following procedures are applications of a Clock
    Clock::{
    public:
        Time<<>>; //elements of unnamed object can be referenced directly

        proc Second ();
        proc Minute (){for(seconds = 0; seconds < 60; seconds++)     Second();}
        proc Hour   (){for(minutes = 0; minutes < 60; minutes++) Minute());}
        proc Day    (){for(hours   = 0; hours   < 24;   hours++) Hour();}
        proc Month(integer ndays){for(days=0; days < ndays; days++) Day();};
        proc Year(bool leap_year = false);
        proc Era()    {for(years = 0;;years++)  Year(years / 4 * 4 == years)}
        proc count()  {Era();}              //the counting procedure
        proc stay();                        //the staying (non-counting) procedure
        proc reset()                        //resets time to null
        proc stop();                        //the stopping procedure
    }

    proc Clock::Year(bool leap_year = false)
    {
        months = 1;     Month January(31);
        months = 2;     Month February(leap_year ? 29 : 28);
        months = 3;     Month March(31);
        months = 4;     Month April(30);
        months = 5;     Month May(31);
        months = 6;     Month June(30);
        months = 7;     Month July(31);
        months = 8;     Month August(31);
        months = 9;     Month September(30);
        months = 10;    Month October(31);
        months = 11;    Month November(30);
        months = 12;    Month December(31);
    };
```

The event `Ptime` represents the passing time, which lasts the same time span as the attribute `occurred`.

```
class Ptime
{
publiic:
   Time ptime;
   extern Clock clk;
   Ptime(event ev_begin, event ev_end)
   {
         Time t1 = clk.count()@[it at
         ev_begin);
         Time t2 = clk.count()@[it at
         ev_end;];
         ptime = t2-t1;
   }
   Ptime(event occurred)
   {
         Time t1 = clk.count()@[it at
         occurred@start);
         Time t2 = clk.count()@[it at
```

```
            occurred@finish;];
            ptime = t2-t1;
    }
}
```

The class Ctime represents the current date-time, which has passed since the start of A.D.

```
event first_day_of_A_D;
//the first day of A.D.
class Ctime
{
 public:
   Time ctime;
//Ctime is the current date-time shown
by some clock
   Ctime(event ev)
   {
         ctime = $Clock$.count()@[it
at ev@start]-
         $Clock$.count()@[it at first_day
         _of_ A_D@start];
   }
}
```

The class age represents the age of an object at a query-point.

```
class age Time.years
[
   Object   aged;
//the object whose age is queried
   Ctime    querypoint;
//the time of the query
   Ptime    pt(aged@creation),querypoint);
   it == pt@years;
]

I walk(),during(Ptime(.hour(3)));
//I am walking during three hours
I walk@[during(Ptime(.hours=3))];
//the same
I walk()@start,at(Ctime(13,15,15,11,2003));
//I started to work at 13:15 15/11/2003
I walk(),finish@[at(Ctime(13,15))];
//the same as before
```

6. The way of *T*

The representation of elementary constructs does not necessarily mean that a KR language is also able to represent really complex knowledge structures like complex events, dynamic and temporal systems, behaviors, general situations concerning crisis management, terrorism, heritage etc. All KR languages developed up to now have failed in complex representations, but T has the potential to avoid the same fate due to the superior combination of a C++-like notation and narrative specifications similar to narrative texts in human languages.

The limited size of this article does not provide the ultimate proof of the expression power of ***T***. Hence the emphasis of this section will be on those characteristics of ***T*** and C++, which are decisive for producing complex representations.

6.1 The real expression power of C++

C++ (actually C/C++) is not just another programming language. In reality, it is a **universal programming language**, which has already expunged all other compiled languages, leaving space only for those interpreted as Java, C#, JavaScript, PHP, Perl, Basic. The most expressive of the latter are also based on its conceptual system, thereby providing more proof of its universality. The language constitutes the base of virtually all sophisticated software such as operating systems, programming software (compilers, interpreters, linkers, debuggers) as well as complex application software for various purposes. So far, no programming tasks have been discovered that were impossible (or nearly impossible) to compose in C++ due to the latter's restrictions.

The huge expression power of C++ is by no means accidental. It has been long since forgotten that software systems with a single compiled language were absolutely unimaginable in the first decades of high-level programming. C++ is the ultimate result of a three-decade-long process, in which more than two thousand programming languages have been created. More on this theme can be found at Ref.[5] Chapter 2.

The root of the expression power of C/C++ lays in the unlimited scalability of its applications. The language can be used to produce a simple program implementing simple graphical operations like drawing a line on a screen. This program can be subsequently extended into the set of graphical classes of arbitrary complexity. The same is also possible for applications of any kind and purpose.

In contrast to developers of other programming languages, developers in C++ do not have to worry about competing with these or other language restrictions limiting the size and complexity of their developments. The real limits to C/C++ programmers are those of a particular solution. Once a certain solution fails to work because of (e.g.) too much input data, it can be replaced by another solution, which can do the job. Thanks to the superior expression power of C++, such a change will never require moving it to another programming language.

The complete knowledge representing the behavior of a complex programmed system is pretty close to knowledge already represented by its software. Just consider a dogfight between automated flying drones, both requiring and implementing extensive software packages that control every aspect of the drone's behavior such as maneuvering, monitoring its current status, tasks to be achieved and so on. Since a drone has a lot of sensors, the software must be able to evaluate the drone's environment including not only wind speed and air temperature but also the functionality of other drones, the terrain below, the distance it can fly with its current supply of fuel and so on.

The non-executable specification of the drone's fight scenario will once again define all of the aforementioned entities, most of them with more detailed specifications. It will also add some other entities such as maintenance schedules, its storage, prelaunch preparation, transportation and so on. For such cases, ***T*** also can be used as a specification language depicting the complete set of involved entities. A part of these entities could be used later directly by the software.

6.2 Expressing non-executable code in *T*

The general procedure for expressing knowledge in ***T*** is similar to that of expressing knowledge in a human language. A complex specification (e.g., a crisis management scenario) can be translated sentence after sentence into ***T*** with subsequent optimization consisting of removing duplicate and superfluous references, repeated specifications and so on. Because ***T*** uses commonsense ontologies of time, event and space such a translation is always possible.

A sentence constructed in a human language can be translated into one or several ***T*** constructs. Below is the translation of the sentence "*Pete came from the training at three o'clock, he looked exhausted*". Please note that the translation to ***T*** converts the previously implicit context into explicit representations. Thus the phrase "he looked exhausted" means that there is someone (e.g., the author of this sentence) who has seen Pete. Another explicated entity is the place where Pete arrived.

```
Human       Pete,
                thirdPerson;
Place       arrival_place;
info        Human::evaluate{exhaust, fresh,
normal}(Human estimated)
Activity    training; //Activity has the
attribute Place  referred below
Object way(Place from, Place to);
proc Human::come(Place ap);

Pete::come(arrival_place);
_v at 15.00;
_v was;
_v.ap@from(training@place);
thirdPerson::evaluate.exhaust(Pete);
```

Last five lines can be expressed in a single ***T*** sentence.

```
Pete::come(arrival_place), at 15.00,
was,
from(training@Place),
thirdPerson::evaluate.exhaust(_n);
```

When a knowledge system uses alternative paradigms of basic notions (e.g., its own time paradigm), such a paradigm can be defined with the help of conventional classes and attributes. ***T*** completely supports the classical C++ freedom in extending and changing the designed code.

6.3 The lexical content of *T*

In order to specify some notion, e.g. "terrorism" one has to express it in ***T*** in the same way in which the same notion is expressed by conventional human languages. For developed human languages equipped with huge vocabularies all one needs to do is to search for the word in a dictionary. An Englishman striving to convey the word "terrorism" to its German interlocutor can find its German equivalent - "Terrorismus" in a bilingual dictionary. A person wanting to understand the meaning of this term looks through a standard language-appropriate dictionary for a definition and finds something like "systematic use of violence and intimidation to achieve some goal".

Because T currently possesses no dictionary, a notion has to be defined from scratch. This is a tremendous amount of work since it means many other notions, like "human", "society", "behavior", "fight" also have to be defined. Defining these notions in ***T*** will probably require producing a dictionary with several thousand entries. The work, however, only needs to be done once; all subsequent definitions will reuse this initial content.

6.4 Implementations

Theoretically, ***T*** can be used in the same way as a human language for communicating between people for any reason but this hardly makes any sense. The purpose of ***T*** consists of implementing different KR-related tasks. The most obvious implementation of ***T*** consists in creating the knowledge database, which can be filled by various specifications produced either automatically or manually.

Another implementation could be a translator from a human language (English) to ***T***. ***T*** can be used for the formal documenting of arbitrary systems. ***T*** allows existing programming languages to be converted into T subsets, which are functionally equivalent to executable subsets of English

(German, French, etc.) applied in particular imperative communications. The first step of such a conversion requires the explicit expression of the communication context, which in the case of a programming language is its innate processor. The second step consists of replacing the original set of linguistic items of a programming language, with its ***T*** counterparts built on the basis of the previously produced specification in ***T***.

Changing the grammar does not influence the semantics of a programming language. It is like creating an interface for a library of functions implemented in one programming language but intended for use in another (e.g., a library implemented in C needs COBOL's interface for use in COBOL). Applying such a conversion to all languages used on any computer system allows for the creation of a monolingual communication environment in which ***T*** can be used as the only language applied in all communication occurring in this system. Monolingual systems make a computing environment friendlier, allowing a user to control all system features with only a set of browse and development tools. Such a system is easy to use and very easy to extend. Also, it has no semantic gaps characteristic of the traditional specifications of programming languages. The example for the specification of the innate processor of C++ can be found in Ref.[5] Chapter 5.

6.5 The current state of *T*

The language was originally defined in Ref.[23] Currently, it is in a stable form but still without a dictionary. More language examples can be found in Ref.[5] Chapter 7.

As a formal equivalent of a natural language, ***T*** has no predefined computer implementation. Hence, the current development is concentrated on the thorough definition of its syntax and semantics, which requires composing various examples in ***T*** code. Any viable software implementations can only be done after finishing the current development stage.

7. Conclusion

1) There are two completely different KRs – one grounded in scientifically proven principles but failing in practical application and one based on the concepts of mainstream programming and demonstrating efficiency unachievable by the former. The semantic gap between these KRs was caused by the historical failure of computer science to produce the adequate definition of a programming language. While it is impossible to conceptualize an undefined thing, the mainstream programming languages were practically excluded from the conceptual world of computer science.

2) The failure in defining a programming language was the result of ideological preferences established in the early years of computer science. All non-mathematical views of a programming language were routinely rejected and the adequate definition was among them. Historically, it was one of the first programming language definitions ever. According to it, a programming language was viewed as a conglomeration consisting of a genuine language (a system of lexical items) and an execution unit controlled by commands formulated in this language.

3) An adequate definition opens the way to building a universal representation language, because it allows a direct comparison between the natural languages representing the genuine knowledge structures and the programming language being very effective in producing the binary models thereof. According to the definition presented in this article, the language part of a programming language relates to a universal representation language in the same way in which a jargon (a subset of a natural language restricted to particular communicators and a topic) relates to its full-fledged superset. Henceforth the task of building a universal representation language is tantamount to the task of extending some jargon to a full-fledged natural language.

4) A universal representation language is a full-fledged formal language, which is not restricted by whatever predefined communicators or communicated themes and can be used for expressing any information ever formulated in any human or computer language. Theoretically, there is no restriction on the number of full-fledged formal languages exactly as there is no restriction on the number of natural languages. However, since each of them will ultimately possess the exact same expression power, one full-fledged formal language is completely sufficient. Taking into account different expression abilities of distinct programming languages, the best match as the extension basis makes the most powerful of all programming languages – C++.

5) The universal representation language ***T***, built on the basis of C++ is the set of linguistic items employed in the manner of a natural language with the purpose of information exchange between various communicators. The language is not confined to any particular representation domain, implementation, communicator or discourse type. Assuming there is sufficient vocabulary, each text composed in any of the human languages can be adequately translated to ***T*** in the same way as it can be translated to another human language. ***T*** allows the creation of a monolingual communication environment in which it could be used for expressing miscellaneous imperative and indicative code.

6) The semantics transmitted by ***T*** code consist of conventional knowledge regarding objects, actions, properties, states

and so on. According to the presented approach, notions based on common knowledge constitute the fundament of all philosophical and scientific conceptual systems allowing the representation of all these systems in the form of class hierarchies.

REFERENCES

[1] Grimm S, Hitzler P, Abecker A. Knowledge Representation and Ontologies. in Semantic Web Services, R. Studer, S. Grimm, and A. Abecker, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg; 2007. pp. 51-106.

[2] Hoekstra R. Knowledge Representation. in Ontology representation design patterns and ontologies that make sense, Amsterdam; Washington, DC: IOS Press; 2009.

[3] Ellis MA, Stroustrup B. The Annotated C++ Reference Manual. Reading: Addison-Wesley Professional; 1990.

[4] Sunik B. Theory of meaningful information, background, excerpts and meaning representation. Artificial Intelligence Research. 2013 May; 2(3): 102-122. https://doi.org/10.5430/air.v2n3p102

[5] Sunik B. Theory of Meaningful Information. 2018.

[6] Chappell T. Plato on Knowledge in the Theaetetus. The Stanford Encyclopedia of Philosophy. 2013.

[7] Carpenter AN. Kant, the Body, and Knowledge, presented at the 20th World Congress of Philosophy, Boston, Massachusetts; 1998.

[8] Sowa JF. Knowledge Representation: Logical, Philosophical and Computational Foundations. Pacific Grove, CA, USA: Brooks/Cole Publishing Co.; 2000.

[9] Backus JW. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference, in ICIP Proceedings, Paris; 1959.

[10] Sammet JE. Programming languages: history and fundamentals. Prentice-Hall; 1969.

[11] Sebesta RW. Concepts of Programming Languages, 2nd ed. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc.; 1993.

[12] Pratt TW, Zelkowitz MV. Programming Languages: Design and Implementation, 4th edition. Upper Saddle River, NJ: Prentice Hall; 2000.

[13] Gorn S. On the logical design of formal mixed languages. 1959. p. 1-2.

[14] Chu Y. Direct Execution In A High-Level Computer Architecture. 1978. p. 289-300.

[15] Sunik B. The paradigm of OC++. SIGPLAN Notices. 2003; 6: 50-59. https://doi.org/10.1145/885638.885648

[16] Britannica Encyclopedia Ultimate Reference Suite. 2009.

[17] Sunik B. The Ultimate Representation of C++ Semantics. Theory of Meaningful Information[Internet]. 2012. Available from: http://generalinformationtheory.com/cpp/cpp2.php [Accessed: 15-Mar-2015].

[18] Smith M, Welty C, McGuiness D. OWL Web Ontology Language Guide[Internet]. W3C; 2004. Available from: https://www.w3.org/TR/2004/REC-owl-guide-20040210/. [Accessed: 04-Mar-2018].

[19] Clark P, Porter B. KM - The Knowledge Machine 1.4.0: Reference Manual. 1999. [Internet]. Available from: http://www.cs.utexas.edu/users/mfkb/manuals/refman.pdf. [Accessed: 13-Mar-2015].

[20] Oaklander LN. The Ontology of Time. Amherst, N.Y.: Prometheus Books; 2004.

[21] Akerkar R. Foundations of the Semantic Web: XML, RDF and ontology. Oxford, U.K.: Alpha Science International; 2009.

[22] Curé O, Blin G. RDF database systems: triples storage and SPARQL query processing. Amsterdam; Boston: Morgan Kaufmann; 2015.

[23] Sunik B. The language T. SIGPLAN Notices. 2005; 5: 28-38. https://doi.org/10.1145/1071221.1071224