Natural Language of Application Domains versus Domain Specific Programming Languages

Cuong Bui Donald Ephraim Curtis Teodor Rus

Department of Computer Science
University of Iowa
Iowa City, IA, USA
{ckbui,dcurtis,rus}@cs.uiowa.edu

Abstract

Research destined to make programming easier [SC98, Har08, May] and to enrich computer application domains (ADs) with computational thinking developed by information technology (IT) [Win06, TW07] are hot topics of today's research. With the project on the application driven software development [RC06, RC07] we address the same problem seen from the perspective of the computer user. What we see as the problem is not the difficulty of programming. The problem seems to be the computer-based problem-solving technology which, unlike other technologies, requires computer user to be a programmer, that is, to be a computer expert. We believe programming can be made easier for the computer user by developing software tools that support computer use without programming. The principle we follow is "take the computer to the problem rather than taking the problem to the computer". This is achieved by changes in problem solving methodology which affect both the AD and the IT. With our tools, AD experts develop problem-solving algorithms and express them in the natural language of the domain while IT experts develop and implement programs using programming languages. Hence, programming is done by computer experts and it is easy because it belongs to their expertise; problem solving algorithms are developed by problem domain experts using the natural language of the problem domain and it is easy because it belongs to their expertise. This leaves room for the computer scientists to develop problem-domain creativity support tools [Shn07] which then contribute to accelerating discovery and innovation.

Our approach is based on a dramatic change of how language is used in computer-based problem solving. Domain specific languages are programming languages tailored to particular domains for particular patterns of program creation [vDKV]. The natural languages of a computer application domain is a conceptual languages used to communicate ideas between domain experts. Regardless of the level of abstraction, a domain specific language represents code to be performed by the machine while the natural language of an application domain represents domain concepts to be handled by the domain expert using domain logic. At this level it is irrelevant whether or not computations implied in these abstractions are performed by an expert's brain or by a computer used by the expert. The purpose of the application driven software we are developing is to allow problem domain experts to develop problem solving algorithms using the natural language of the domain and execute these algorithms in the domain (not in the machine environment) thus liberating the computer user from programming[Har08].

The goal of this paper is to demonstrate how language is used in our approach of computer-based problem solving methodology. We illustrate it on problems from such different computer application domains as high-school algebra and natural language processing. To prevent the reader from being trapped into the conventional thinking pattern about languages we dedicate the core of this paper to contrasting the concepts of "natural language of the application domain" and "domain specific language".

1 Introduction

The computer is a wonderful problem-solving tool that in many ways can do computations faster and more reliably than the brain. But in the end it is just a tool that can help people solve problems providing that people learn how to use it. The biggest hurdle people face when using computers to solve their problems is determined by the current computer-based problem solving methodology which requires computer user two perform in two steps:

1. Develop a conceptual model of the problem and its solution algorithm [1]
2. Map the conceptual model and solution algorithms into a program in computer memory.

Program execution is then carried out by the computer that, for the purpose of this paper, can be seen as a tuple $\langle PC, Execute(), Next()\rangle$ that performs as follows:

1. PC is a program counter register that points to the memory location that holds the operation to be executed;
2. Perform() is the wired action carried out by the machine while performing the operation shown by PC;
3. Next() is the wired action that uses current instruction to select the next instruction to be executed by the computer.

. The process by which machine instructions are carried out by the computer while executing a program is further referred to as the program execution loop (PEL):

```
PC = First(Instruction);
while(not halt):
    Execute(PC);
    PC = Next(PC);
```

Irrespective of its level of abstraction the program represents sequence of machine instructions, encoded as bits and bytes, that "instruct" the computer what to do. On the other hand the conceptual model represents concepts, i.e., problem domain abstractions fabricated by people's mind. Because the essence of programming is program creation while the essence of problem solving is conceptual model development, this gap is irreconcilable. It is the cause of the difficulties encountered by computer user. Depending upon computer user expertise, computer programming could be a nightmare (for a non expert) or a delight (for computer experts). Software technology was developed by computer experts to make computer programming easier for other computer users. Software technology helps computer user map her problem solving algorithms into programs, but software technology does not liberate computer user from (the nightmare of) programming.

There are two issues we see with current software technology. The first is that computer technology is aimed at making programming easier for the experts of possibly infinite number of computer application domains, each with possibly infinite many problems. As a result we are looking at increasing *software tool complexity* at a level which threatens to *kill the information technology* [Kri]. The second is that as the complexity of software tools increases the complexity of tool usage and maintenance also increases, and threatens to destroy computer usage [IBM]. The question is then how can we make computer use easier for ever increasing number of computer users without creating more complexity. We believe that the solution is to develop software tools that liberate the computer user from programming. That is, we would like to create a computer-based problem solving methodology which does not require computer user to map her problem model and solution algorithm into a program. This cannot be achieved by outsourcing the mapping of program model and solution algorithm into a program because for that, the domain expert would need to specify the mapping which may encounter even more difficulties than those involved in programming.

Addressing the issue of *software tool complexity* above we observe that current computer-based problem solving methodology makes no distinction between various problem domains. But experts of different problem domains may use different mechanisms for the development of their problem model and algorithms. So,in

[1] We use here the term algorithm with an intuitive well-defined meaning, which may depend upon the problem domain

order to proceed with computer user liberation from programming we need first to remove the current problem solving pattern that puts problem solving using a computer into a "one-size-fits-all" pattern. This gives us the idea to develop a computer-based problem-solving methodology where each problem domain may have its own computer-based problem solving pattern. But this means that we actually want to follow the principle of "taking the computer to the problem rather than taking the problem to the computer". This sounds quite nice, but how to achieve it? The solution is neither simple nor easy to implement. It cannot be carried by problem domain expert or by computer expert independent of each other. It requires a cooperation between them which can go on the following lines:

1. Problem domain expert characterizes her application domain in terms of well-defined concepts, that have computational meaning and (a) are universal over the problem domain, (b) are stand-alone, and (c) are composable. This allows her to outsource the problem of creating computer artifacts (including programs that implement these concepts) to the computer expert.

2. Computer expert develop implementations for the concepts that characterize the application domain and broadcast them to the domain using appropriate URI [BLFIM98].

3. Problem domain expert and computer expert structure the application domain using an application domain ontology where concepts are associated with the URIs of the computer artifacts implementing them.

We call the activities (1),(2),(3) above the Computational Emancipation of Application Domain (CEAD). CEAD can also be seen as the practical implementation of the *computational thinking* [TW07]. Now the process of computer-based problem solving in a CEAD-ed domain may be further shared by the domain expert and computer expert by the following protocol:

1. The domain experts collaborate (again) with the computer expert to develop a domain dedicated language to be used by the domain expert to develop problem models and solution algorithms. We call this language the *natural language of the application domain*, NLD. We discuss this language in section 2 of the paper.

2. Computer expert creates a virtual machine dedicated to the application domain that can execute (by interpretation or by translation) the domain algorithms in the domain environment not in the computer environment. This is what we mean by "take computer to the problem rather than taking the problem to the computer". This is discussed in section 3 of the paper.

3. Domain expert and computer expert collaborate for the development of software tools that optimize and automate the problem solving process. These are creativity support tools [Shn07] that support NLD evolution with the domain ontology thus accelerating the domain discovery and innovation. Some of these tools are discussed in section 4.

1.1 Related work

The entire research on programming is actually dedicated to making programming easier and consequently is related to our work. Therefore we need to be very specific in identifying the differences we make. For that we start with the observation that programming per say is not a goal, it is a mean to achieve the goal. The goal is problem solving. Of course, program creation is by itself a problem to solve and thus a goal which has a special place in the hierarchy of goals called *computer-based problem solving process*. The link which binds together the entire research on making programming easier is the process of *raising the abstraction level of program notation to match the programmer interest and problem to solve*. The raising of the abstraction level led from machine code to assembly, Fortran, Cobol, and further to Object Oriented. The matching of the problem to solve led to Fortran for easy evaluation of numerical formals, to Cobol for easy data manipulation, to Object Orientation to increase programming productivity, and further to today Domain Specific Languages [vDKV, MJS05] *to make it easier to develop and implement programs*. But in all these situations the abstractions represented by the languages thus created are machine codes. We may observe that though the problem domain plays a special role in the development of domain specific languages, programming was kept universal. That is, the abstractions that makes it easier to map domain

specific concepts into machine representations are not abstractions that would free the computer user from the dependence of computer infrastructure. By the contrary they strengthen this dependency.

Our main contribution is then the development of a methodology that bring computer to the problem domain not problem domain to the computer. We advocate the development of abstractions that liberate computer user from programming. This means that we advocate the creation of languages dedicated to problem solving not to program development. Of course, if the problem to solve is program development then the language we advocate is dedicated to program development. For that to become reality we notice that different problem domains may use different mechanisms for modeling their problem solving process. Therefore we develop a mechanism that lets computer user develop her problem domain specific language, which actually coincides with the natural language used by domain experts to formulate problems, develop solution algorithms, and to communicate with each other during problem solving process. This is why we call such a language the Natural Language of the Application Domain, NLD. With the development of Meta Programming System, Dimitriev [Dim] has shown a practical implementation for this methodology, in the framework of today computer technology, for the problem domain identified as the *program creation*. With the development of the special train set domain, Smith, Cypher, and Schmucker [SCS96] have show a practical solution to this methodology with the problems raised by children game Cocoa. With the research we reported in [RC06, RC07] we have shown that this is practically feasible within the framework of current technology for any CEAD-ed problem domain, i.e., for any *computationally emancipated application domain*. There may be many other achievements toward the liberation of computer user from programming which we are not aware of.

To prevent the reader from being trapped into conventional thinking about languages, we summarize below the main differences between the terms *domain-specific language* (DSL), as used in current computer technology, and the term *natural language of the domain* (NLD), we use in this research:

- A DSL is a programming language dedicated to a specific domain. Hence, its goal is to make it easier to map domain concepts into machine representations; NLD is a natural language used by domain experts to model problems and solution algorithms using the concepts of their domain of expertise.
- The user of a DSL is heavily dependent on programming infrastructure on which programs thus created need to run; the user of a NLD depends only on her domain of expertise.
- The DSL (as any programming language) is frozen. It evolves only by re-design and re-implementation; NLD is freely evolved by the domain expert with the knowledge she acquires during cognition process.

2 Natural Language of the Application Domain

The problem solving paradigm we advocate in this paper liberates computer user from programming, but it does not liberate her from solving her problems. That is, with this approach of problem solving AD expert formulate the problem and develop solution algorithms using domain specific concepts. Therefore we call the language an AD expert uses the *Natural Language of the Domain*, (NLD). The vocabulary of an NLD is well defined by the collection of terms used in the domain ontology to denote domain concepts enriched with computer-technology terms whose usage transcended from programming language to natural language preserving their meanings through the computational thinking process[Win06].

The meaning of the terms used in the domain ontology is informally specified as the *domain abstractions that domain experts denote by these terms*. Formally, these terms represent domain concepts that have a computational meaning which is: universal in the domain, stand-alone, and composable. For example, in high-school algebra, integer addition, denoted $+ : I \times I \to I$, is:

1. universal over the set I of integer numbers because $n_1 + n_2 \in I$ for any $n_1, n_2 \in I$,
2. stand-alone because the computation of the number $n_1 + n_2$ depends only of the numbers n_1, n_2,
3. composable because it can be composed with other operations, i.e., $\forall n_1, n_2, n_3 \in I$, $(n_1 + n_2) + n_3 \in I$, that is $+$ is the same operation, regardless of how do we define its arguments.

These kind of abstractions are well-known in mathematics but are not always preserved in programming languages. For example, a standard library of a given programming language PL contains functions denoting computations that may be universal with respect to PL, but usually are not stand-alone because they can be used only in the context of a valid program of PL, and are composable only through the composition pattern defined by the PL. Hence, these abstractions do not characterize the domain; they characterize the machine-computation that abstractions of the PL represent. Since the NLD is a natural language we requires its abstractions to be universal, stand-alone, and composable over the domain they characterize.

A less trivial example of NLD terms satisfying these properties is provided by the terms used in a natural language processing domain. Figure 1 shows an ontology of a subset of the natural language processing domain that deals with the problem known as the *recognizing textual entailment*[BM05] further referenced as the RTE problem, which is stated as follows: *Given two text phrases, T and H, and a knowledge data base such as* **WORD-NET**, *develop an algorithm that decides whether T implies H or not.*

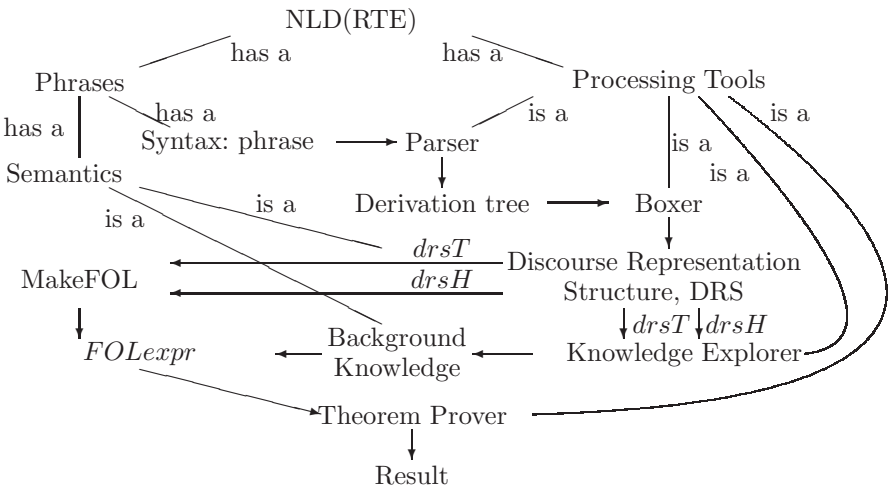

Figure 1: An ontology of the RTE problem

Notation: we use the following notation in Figure 1:

1. lines denote `is a` or `has a` relationships between concepts;
2. arrows entering a concept node denote the input to the computer artifact implementing the concept labeled by that node;
3. arrows exiting a concept node denote the output of the computation performed by the computer artifact labeled by that node.
4. an ontology of the First Order Logic (FOL) is imported in the RTE ontology.

The concepts used in the ontology in Figure 1 such as Phrase, Syntax Structure, Derivation tree, Background Knowledge, Meaning, First Order Logic (FOL) Expression, Discourse Representation Structure, etc., are all universal because they denote mathematically well-defined abstractions and all natural language experts use them with the same meaning. The concepts such as as Parser, Boxer, Knowledge Explorer and Theorem Prover, in addition of being universal in natural language domain, are stand-alone because the performance of the computations they represent depends only on the arguments they take in order to perform their tasks. Further, they are also composable because the computations they perform can be composed with one another according to their input/output behavior patterns, thus defining larger computations that represent valid concepts in the natural language of the domain. For example, Parser, Boxer, Knowledge Explorer, and Theorem Prover can be composed to form a Textual Entailment Recognizer.

Further, in this ontology the concept represented by the term *Theorem Prover* is universal, stand-alone and composable because it has as the meaning a computation that for any logical expression of the form $T \rightarrow H$ it returns the truth-value of the logical implication $T \rightarrow H$.

For any domain, the domain ontology is ever-expanding, that is, it is built up "interactively and inter-subjectively" by the language used to communicate between domain experts based on their "commonsense" ontology [Bat93]. New terms of the language come from interaction between domain experts or are inherited from other domains. For example, the RTE domain contains the terms: *phrase, grammar, derivation tree, discourse representation structure* which denote concepts created in the RTE domain while the term *theorem prover* is inherited from mathematical logic, and terms such as *assignment, variable, if, then, else* are inherited from computer technology. Usually the terms inherited by a domain from another domains may exists already in the domain that inherits them with a different semantics. However, here we assume that terms in NLD inherited from computer technology (through computational thinking process) are used in the NLD with the meaning they have in the computer technology. In addition, we assume that the domain expert, during problem solving process, may create new terms as she needs them for the development of problem model and solution algorithm.

Domain experts use NLD to express their ideas. They still need to learn some syntax in order to express these ideas. This is facilitated by the fact that they use *the syntax of their natural language*. The meaning of their communication is specified by the domain ontology and therefore the semantics of NLD is dynamic and is totally controlled by the domain experts. But during problem solving process the domain experts use the NLD to communicate with the IT experts as well, whose Natural Language of the Domain are programming languages (including DSL). How can this be done without asking AD experts to learn PL or IT experts to learn NLD? The answer to this question is provided by the process of *Computational Emancipation of the Application Domains* (CEAD). Since domain concepts are universal over the domain, are stand-alone, and are composable, they are implemented by the IT experts as computation-artifacts on computer platforms of choice and are associated with the terms of the ontology using URI-s. This allows domain experts and computer experts to bridge the gap between natural language and computer language. Examples of such implementations would be libraries of functions that are universal (i.e., can be called from any programming environment, are stand-alone (i.e., their execution depend only on their arguments), and are composable (i.e., they can be freely composed with other such functions). We believe that current function-libraries developed for given programming language environments could be easily transformed into functions satisfying these properties. The consequence is that with a CEAD-ed domain, the domain experts can communicate among them and with the IT experts using the terms of the ontology because the communication gap is bridged by the URI of the computer-artifacts associated with the terms of the ontology.

In order to keep NLD simple, as simple as possible, the above informal discussion can be formalized by the following three layers of NLD specification:

1. **Vocabulary:** the vocabulary (or lexicon) of the NLD is a finite set of terms $V_D \cup A_D$ where V_D is the set of terms used to denote concepts in the domain ontology and A_D is the set of terms inherited from technology or invented by domain expert to express *actions* used to develop problem models and solution algorithms. For example, the term *assignment* (inherited as the symbol :=) is used to initiate the computation (or the evaluation) of a domain concept and to give the result as the value to a domain variable (unknown). Semantically every term of the vocabulary is associated with three properties: arity, signature, and type:

   (a) the *arity* of a terms t shows the number of arguments it takes in order to express and action;

   (b) the *signature* of a term t shows the order and the type of the arguments it takes in order to express an action;

   (c) the *type* of a term t is the type of the concept created by the action it performs.

   Note: if t is a term in the vocabulary and its arity is 0 (zero) then it represents a concept in the ontology and the action it performs is well defined by the URI associated with it.

2. **Simple phrase:** a simple phrase of NLD is any action specified by the sequence $t_0\ t_1\ \ldots\ t_k$ where $arity(t_0) = k$, $sig(t_0) = t_1, t_2, \ldots, t_k$, $type(t_0\ t_1\ \ldots\ t_k) = type(t_0)$. For notational purposes the term t_0

representing the action may be freely distributed over it arguments as is the case of $if, then, else$ inherited from programming languages, or of the parentheses inherited from mathematics. If $arity(t_0) = 0$ the simple phrase is a concept in the ontology. Semantically a simple phrase represents a unit of action employed by domain expert in order to denote a step of her solution algorithm. For example, $v := n_1 + n_2$ denotes the action of adding two numbers and giving the resulting number as the value to the variable v.

3. **Phrase:** a phrase in NLD is either a simple phrase or an action $t_0\ t_1\ \ldots\ t_k$ where $arity(t_0) = k$, $k \geq 1$, and $t_1, \ldots t_k$ are phrases. Semantically a phrase is a composed action. That is, a phrase represents a solution algorithm.

Example: The NLD algorithm that solves RTE problem follows:

```
RTEdecider:
   Input: T, H : Phrase; Output: Result: Phrase;
   Local: treeT, treeH: Derivation Tree;
          drsT, drsH: Discourse Representation Structure;
          bk,ET,EH: FOL Expression;
          do:
             treeT = Parser(T); treeH = Parser(H);
             drsT  = Boxer(treeT); drsH = Boxer(treeH);
             bk    = Knowledge Explorer (drsT,drsH);
             ET =  MakeFOL(drsT); EH = MakeFOL(drsH);
             Result = TheoremProver((bk and ET) implies EH))
```

A natural language expert can read and understand this algorithm without any other explanation because the concepts used are those in the RTE ontology. Moreover, the language expert understand the computation performed by the algorithm without thinking at the machine that may execute it because each of the concepts used in the algorithm has well-defined meaning for her at the natural language level. The computation expressed by the algorithm is conceptually carried out by the domain expert handling the algorithm. This is feasible because every term of the algorithm is well-defined for the NLD expert. Therefore, to generates the `Result` the domain expert rely only on her domain knowledge. To carry out this computation she may use the virtual machine dedicated to RTE problem, as we will explain in Section 3, though she does not need to know where and how is this computation physically done.

3 Executing Natural Language Algorithms

Natural language algorithms are executed by a Virtual Machine (VM) dedicated to the domain. This machine is an abstraction that consists of: a *Concept Counter* (CC), an *Abstract Processor* (AP), and a mechanism called *Next()*. Given a CEAD-ed domain ontology (DO) the VM(DO) performs as follows:

1. CC points to a concept in the DO.
2. AP executes the computation (if any) associated with the concept shown by CC.
3. Next(CC) determines the next concept of the DO to be performed by the machine.

In comparison with the behavior of a universal computer described by the PEL-loop in Section 1, we denote the behavior of the VM(DO) by Domain Execution Loop (DEL) and describe it by the following pseudo-code:

```
CC = StartConcept(DO);
while(CC not END):
    Execute(AP,CC);
    CC = Next(CC);
```

The DEL we describe mimics the behavior of the PEL-loop performed by real computers, but there are major differences. The Concept Counter (CC) of the DEL-loop is similar to the program counter (PC) of the PEL-loop in that it keeps track of what is to be performed. But note that rather than pointing to memory containing machine instructions, the CC points to concepts on the emancipated domain ontology. Additionally, the instruction that PC points to is carried out by a central processing unit as part of the machine hardware. In the DEL loop, the concept pointed to by CC is evaluated by the abstract processor that performs the action Execute(AP,CC). By computational emancipation of the application domain, the concept shown by CC in the ontology may be associated with a stand-alone computer artifact identified by an URI. In this case Execute(AP,CC) creates a computer process and instructs it to execute the computation-artifact that is associated with the concept pointed to by CC. Thus, we must emphasized that Execute(AP,CC) is *not* carrying out machine instructions but is performing computation, if any, that is associated with CC. Machine instructions involved in this computation, if any, are performed by the computer process generated by Execute(AP,CC). Hence, from a domain expert viewpoint Execute(AP,CC) is actually a computational process performed by the brain, with or without the assistance of a real computer platform. The Next() component of the virtual machine is responsible for selecting the next component to be executed by the virtual machine. This is similar to the circuitry of the real machine which is responsible for selecting the next instruction of a machine-language program. Exactly as in the case of a real computer, we make the assumption that the next-operation performed by VM is *encoded in its current operation*. Therefore, once VM(DO) is started it never halts. This assumption is implemented in VM by the composition operators used to express domain algorithms in terms of domain ontology concepts. Finally, the last similarity we signal pertains to the power usage by the two kind of machines. When the power to the machine carrying out the PEL-loop is switched off, the PEL-loop is broken and the computer becomes useless. The DEL-loop represents the human-cognition process and thus it never halts, unless the human thinking power is cut off.

For an AD expert to solve a problem without programming she follows a four step approach similar to Polya's [Pol73] four steps methodology:

1. Formulate the problem;
2. Develop a domain algorithm that solves the problem;
3. Type the algorithm, i.e., input the algorithm in the computer;
4. Execute the algorithm by a command that sets CC to the first concept to be evaluated by the VM while performing the algorithm.

To demonstrate this approach of problem solving we consider the example of high-school algebra. Suppose that the following assignment is given by the teacher: *students, develop an algorithm to solve the quadratic equation* $ax^2 + bx + c = 0$*, where* a, b, c *are real numbers and* $a \neq 0$*, and then apply your algorithm for the case of* $a = 1, b = 4, c = 4$.

1. The problem is formalized as a formal equality $ax^2+bx+c = 0$ in the language of high-schools -algebra.
2. Using the properties of equality the students develop the solution: $x_{1,2} = \frac{-b+|-\sqrt{b^2-4*a*c}}{2*a}$. Now the students organize computations to be performed by $VM(Algebra)$ as the following domain algorithm:

```
Roots:
  input a, b, c: real where a not zero;
  local t: real;
  out: x1, x2: real;
  t := b^2 - 4*a*c;
  if t >= 0 then
    x1 := (-b + sqrt(t))/2*a;
    x2 := (-b - sqrt(t))/2*a;
  else
    no real solutions
```

3. Input the algorithm, (as a text) in their computers.

4. Initiate the algorithm by typing `Start Roots` and then (at request, perhaps) type:

```
Let a := 1
Let b := 2
Let c := -8
```

In few fractions of second the computer prints

```
x1 = 2;
x2 = -4;
```

There are a few key things to note. The first is that we initiate the virtual machine by giving it the component of the domain algorithm to initiate the computation. In example above this is done by telling the VM to perform "Roots". The concepts used in the domain algorithm, such as $+, -,$ *sqrt* are associated with computer artifacts implementing them on a real computer. The other concepts, such as `Let, if, else, :=, in, out, etc.` are terms inherited from current computer technology in the language of high-school algebra. Students use them because they belong to their natural language and VM understand them by the software supporting this problem solving methodology.

As computer scientists the concepts used in this example are *trivial* because they are also computer operations. But there is a very clear distinction between the two. The algebraic concepts used in the above solution are concepts of high-school algebra and *not* references to the computer functions which perform (approximations) of the computations represented by these concepts. That is, in high-school algebra the operations $+, -,$ *sqrt*, *etc.* are *not* the operations which are utilized by a programming language such as C or Fortran. Rather, these terms represent universal, stand-alone, composable computational processes used by high-school students. In this case, just happens that these symbols represent terms that IT experts are accustomed to seeing in their favorite programming language. With this same idea in mind the reader should observe (again) that the algorithm, along with being expressed in terms of domain concepts, includes *terms inherited into the natural language of the domain from computer technology (computational thinking).* Terms such as *if* (check a property), *else* (a property is not true), and possibly others, refer naturally to the flow of control in the domain algorithm.

This paradigm of algorithm execution does not imply that *computer programming disappears.* It only implies that computer programming is done by professional programmers while AD experts develop and run domain algorithm using domain concepts. Domain experts are not required to be aware of what or where the computer that performs the computation is. This means, students learning algebra may focus on algebra not on Fortran, C, or any other language that may be used to implement the concepts of high-school algebra.

To further the discussion of how expressions in the natural language of the domain are carried out by the virtual machine, and also to show that this paradigm of problem solving is universal (i.e., it is no different when we move from one domain to another), we give another example with a less trivial problem by considering the *recognizing textual entailment problem.* This problem is used in natural language to determine whether one phrase implies another phrase and its RTE solution algorithm was given in Section 2 as the `RTEdecider`. So, suppose we have the phrase `Bill is a man` and the phrase `Bill is human` and we want to check whether `Bill is a man` implies `Bill is human`. Using the `RTEdecider` this is obtained by the following dialog with the machine on which the computationally emancipated RTE domain whose ontology is in Figure 1 is implemented:

% Start RTEdecider:
% Input phrase T: `Bill is a man`
% Input phrase H: `Bill is a human`

After few fractions of a second the computer answer: % True

Note that we use % as a prompt, usual text after the prompt is typed by the machine and `typewriter text` is typed by the user.

As with the high-school algebra example, at the domain level the domain expert does not care how these concepts are performed, just that they are performed and they generate results. But of course, the whole point of this work is to utilize the computer as a tool and the CEADed ontology allows the expert to do that by letting the computer carry out the concepts in the ontology rather than relying on the brain. From the viewpoint of a domain expert using this paradigm of problem solving, the use of the computer is in no way different from using a calculator. The difference is that while a calculator performs simple arithmetical operations, the computer performs complex algorithms associated with concepts in the domain ontology.

4 Optimization of Natural Language Algorithm Execution

Carrying out a domain algorithm incurs the cost of searching the domain ontology for concepts used in the solution algorithm. The challenge to the IT is then to develop software tools that optimize the process of domain execution algorithms. These tools operate on the solution algorithm and domain ontology, collect the ontology concepts involved, perform any validation that may occur, and produce a solution algorithm that eliminates the search process at the algorithm execution time. This optimization can be done by the domain expert "manually", going from concept to concept, collecting the URIs associated with each concept and generating an expression of the domain algorithm using these URIs [RC06, RC07]. This is similar to what a translator of a natural language is performing. Due to the simple structure of the NLD, this can also be carried by a conventional compiler that maps NLD into a process execution language where processes to be executed are completely defined in the ontology. The Software Architecture Description Language (SADL) [RC06, RC07] was created for this purpose. Expressions in SADL contain all the necessary information to carry out the domain solutions without the need to search the domain ontology. Because SADL expressions do not require DEL to search the domain ontology for URIs associated with domain concepts we say they that SADL expressions represent domain algorithms *optimized* for execution in the IT domain. However, SADL is not a programming language and thus it is not a domain specific language either. SADL is a process representation language, tailored to the problem of representing domain algorithms in optimized form which can be carried out by a SADL interpretor using the Domain Execution Loop. This inherently makes the SADL interpretor different from the standard interpreters which generate or perform code based on the expressions given in the language. The SADL interpretor uses the DEL loop to carry out domain algorithms by executing the computer artifacts associated with the domain concepts in the domain ontology. By the assumption of computational emancipation of the domain, these computer artifacts represent computer processes that are universal in the domain, are stand-alone, and composable and thus are easily created and initiated using systems, such as Unix-5, that provide mechanisms supporting process creation, process execution, and controlling process interaction. We illustrate this using the solution algorithm for RTE problem discussed above, whose SADL optimized form follows:

```
<?xml version="1.0" ?>
<sadl>
  <RTE input="URI(T) URI(H)" output="URI(result)">
    <Parser uri="URI(Parser)" input="URI(T)" output="URI(treeT)" />
    <Parser uri="URI(Parser)" input="URI(H)" output="URI(treeH)" />
    <Boxer  uri="URI(Boxer)"  input="URI(treeT)" output="URI(drsT)" />
    <Boxer  uri="URI(Boxer)"  input="URI(treeH)" output="URI(drsH)" />
    <KnowledgeExplorer uri="URI(Knowledge Explorer)"
                       input="URI(drsT) URI(drsH)"  output="URI(bk)" />
    <MakeFOL uri="URI(MakeFOL)" input="URI(drsT)" output="URI(ET)" />
    <MakeFOL uri="URI(MakeFOL)" input="URI(drsH)" output="URI(EH)" />
    <And uri="URI(And)" input="URI(bk) URI(ET)" output="URI(antecedent)" />
    <Implies uri="URI(Implies)" input="URI(antecedent) URI(EH)" output="URI(wff)" />
    <TheoremProver uri="URI(TheoremProver)" input="URI(wff)" output="URI(result)" />
  </RTE>
</sadl>
```

Note: to increase readability here we use the notation "uri(concept)" instead of the real URI of the concept in the ontology.

As observed in previous section, the mapping of the domain algorithm into SADL can be done by the domain expert by hand. This is feasible for toy problems. For more sophisticated problems it is beneficial to automate this process. This is a second class software tools that IT is challenged to develop in order to support computer user liberation from programming. The development of a translator mapping domain algorithms into SADL expressions is facilitated by the following facts:

1. The source language (i.e., the natural language of the domain) is simple. Its vocabulary is defined by the concepts of the ontology associated with computer artifact and the terms inherited from the computer technology. Hence, the semantics of this language is both simple and well-defined. However, notice that this language evolves with the domain expert cognition process.

2. The language is designed to express communication between domain expert and her problem domain and between domain experts of the same domain. Therefore, as we have seen in Section 2, we can design this language to have a very simple syntax, that avoid the usual ambiguities present in natural language. A Generalized Phrase Structure Grammar [KR93] using the rules on pages 33–57, appropriately updated using the informal specification given in Section 2, is sufficient for this purpose. This allows us to use TICS tools[RKS$^+$] to generate automatically the parsers we need and to label the abstract syntax trees generated by these parsers by the URIs of the concepts in the ontology and by the operators encountered in the domain algorithms.

3. The SADL expression is thus automatically generated by walking the abstract syntax tree generated by the parser. No intricacies implied in code generation for actual machines is involved. In addition, we use IPCsem [Rus] to encapsulate the translator into one simple tool that is used by the user with her command: `Map2SADL DomainAlgorithm;`

Since only the vocabularies of different NLD-s are different, the evolving of NLD with the domain cognition process, or the porting of the translator from one NLD to another NLD, can be done by the domain expert by updating NLD vocabulary specification rules and then running the meta-system that generate the translator $NLD \rightarrow SADL$. This follows a similar approach with the MetaWSL [War94] and Meta Programming System [Dim]. TICS tools [RKS$^+$] have been developed for the same goal.

5 Conclusions

Feasibility of computer use without programming is demonstrated by many software tools in use today that expand from using a computer as a typewriter to the enterprise management systems. But there is no recognized trend of liberating computer user from programming. On the other hand the current expansion of computer use (with and without programming) in all aspects of human life cannot be sustained without a serious and systematic research program towards liberating computer user from programming. This does not mean that research on programming would disappear. It only means that computer science community needs to differentiate more carefully between computer users and computer experts and consequently it needs to balance the research on software support for the two classes of experts. Once this realized, it will be easy to observe that software needs of computer users depend on the application domains and domain expertise. In other words, the idea of liberating computer user from programming means actually the liberation of computer technology from the "one-size-fits-all" pattern imposed by current problem solving methodology and consequently we may expect a new explosion of the computer technology.

Why do we want to present this research to AMAST community? The answer to this question resides in the history of AMAST. AMAST has been created to pursue the tendency of using algebraic (formal) thinking as mechanism of discovery and innovation in software development. Twenty some years ago software development had as the objective the development of tools to control the computing power embodied in computer hardware while providing it as services to computer users. Computer was seen as a universal tool that provides universal services. But in the cognition-process spiral *computer technology increases the human cognition power which in turn increases the demand for more computer technology.* This leads to

the diversification of service types different application domains require. Consequently this leads to the need to abandon the "one-size-fits-all" pattern of computer usage. To sustain this aspect of the human cognition-process software tools need to evolve with human cognition process and this can be achieved only by looking at the computer as a cognitive tool, in the hands of computer users. That is, to be further useful, the computer needs to be *a tool in the hands of its users* rather than being *a tool in the hands of its creators*. Due to the thinking inertia created by the successes of computer technology developed so far, we may expect that the main foes of this new paradigm of computer technology are the computer experts themselves. This is why the AMAST following is probably best prepared to understand, to accept, and and to handle this situation. AMAST community evolved as a class of computer experts supporting the setting of software technology on firm, mathematical basis. Now the AMAST-ers are called to collaborate with application domain experts to set computer application domains on firm, mathematical basis, by their computational emancipation, thus contributing to the creation of new software technology meant to the liberation of computer user from programming.

References

[Bat93] John A. Bateman. Ontology construction and natural language. In *Proceedings of the International Workshop on Formal Ontology*, pages 83 – 93, Padova, Italy, March 1993. LABSEB-CNR.

[BLFIM98] T. Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. Rfc 2396: Uniform Resource Identifiers (URI): Generic syntax. http://www.ietf.org/rfc/rfc2396.txt, 1998.

[BM05] Johan Bos and Katja Markert. Recognizing textual entailment with logical inference. In *HLT '05: Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 628–635, Morristown, NJ, USA, 2005. Association for Computational Linguistics.

[Dim] Sergey Dimitriev. Language Oriented Programming: the next programming paradigm. www.onboard.jetBrains.com.

[Har08] D. Harel. Can programming be liberated, period? *Computer*, 41(1):28–37, January 2008.

[IBM] IBM. Autonomic computing. IBM Web Site http://www.research.ibm.com/autonomic/.

[KR93] Hans Kemp and Uwe Reyle. *From Discourse To Logic*. Kluwer Academic Publishers, 1993.

[Kri] P. Krill. Complexity is killing IT, say analysts. Techworld: www.techworld.com.

[May] B.A Mayers. Making programming easier by making it more natural. giove.isti.cnr.it/EUD-NET/slides-workshop/MyersEUP02Italy.ppt.

[MJS05] M. Mernik, Heering J., and A.M. Soane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[Pol73] G. Polya. *How To Solve It*. Princeton University Press, second edition edition, 1973.

[RC06] T. Rus and D.E Curtis. Application driven software development. In *International Conference on Software Engineering Advances, Proceedings*, Tahiti, 2006.

[RC07] T. Rus and D.E Curtis. Toward application driven software technology. In *The 2007 World Congress in Computer Science, Computer Engineering, and Applied Computing, WORLDCOMP'07*, Las Vegas, USA, 2007.

[RKS+] T. Rus, R. Kooima, R. Soricut, S. Munteanu, and J. Hunsaker. TICS: A component based language processing environment. http://www.cs.uiowa.edu/~rus.

[Rus] T. Rus. IPC: An Interactive LR Parser Construction; IPCsem: IPC with semantic functions. Lecture notes on compiler construction. http://www.cs.uiowa.edu/~rus.

[SC98] D. C. Smith and A. Cypher. *Making programming easier for children*. Morgan Kaufmann Series In Interactive Technologies, 1998.

[SCS96] D. C. Smith, A. Cypher, and K. Schmucker. Making programming easier for children. *Interactions archive*, 3:58–67, 1996.

[Shn07] B. Shneiderman. Creativity support tools – accelerating discovery and innovation. *Communications of the ACM*, 50(12):10–31, 2007.

[TW07] S. Tekinay and J. Wing. Cyber-enabled discovery and innovation. *Computing Research News*, 19(5), November 2007.

[vDKV] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. http://www.cwi.nl/arie,paulk,jvisser/.

[War94] Martin P. Ward. Language-oriented programming. *Software — Concepts and Tools*, 15(4):147–161, 1994.

[Win06] J.M. Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006.