

Computer Integration within Problem Solving Process

Teodor Rus ^{a*}

DOI:10.9734/bpi/ratmcs/v8/7105E

Peer-Review History:

This chapter was reviewed by following the Advanced Open Peer Review policy. This chapter was thoroughly checked to prevent plagiarism. As per editorial policy, a minimum of two peer-reviewers reviewed the manuscript. After review and revision of the manuscript, the Book Editor approved the manuscript for final publication. Peer review comments, comments of the editor(s), etc. are available here: <https://peerreviewarchive.com/review-history/7105E>

Abstract

The complexity of current software tools increases with the complexity of problem solving tasks they are designed to assist and are mainly dedicated to computer educated people. On the other hand current computer technology is deeply involved in people's everyday life. This gap deepens and stresses software technology and computer education. By computational emancipation the natural language becomes a family of non-ambiguous languages. This means that every problem solver uses a non-ambiguous natural language, termed here as Domain Algorithmic Language, DAL. Here we show how to develop software tools dedicated to the problem domain and illustrate the methodology we propose with the software tools required by teaching high school algebra.

Keywords: Computer; ontology; problem-solving; problem-domain; software-tool.

1 Preamble

The concept of *computational emancipation of problem domain* has been coined in the paper [1] It denotes the process of a problem domain formalization by means of a *domain ontology*. The goal was the development of software tools that would simplify computer usage in problem solving process and would allow computer users to *computationally emancipate* their problem domains, thus eliminating the fear of jobless society by automation. But these ideas did not reach the targeted computer community. During the last 5 years the usage of ontology as a computation mechanism in software development has taken a different path

^aDepartment of Computer Science, The University of Iowa, Iowa City, USA.

*Corresponding author: E-mail: rus@uiowa.edu;

[2] which does not help making computer usage simpler. In addition, some of the articles in the viewpoint section of Communications of the ACM show that confusion about what is and what is not computer science deepens, and the misunderstanding of artificial intelligence threatens to derail further developments of computer usage as a problem solving tool. Therefore I decided to republish this paper thus hoping that its main ideas will reach computer community. The basis of these ideas is the assumption that *computer is a brain tool* and computer science is simple *the science of computer use during problem solving process*. Problem solving is characteristic to all humans and the human brain is the human organ in charge of performing it. Hence, the computer needs to be seen as a brain assistant and natural language should be the language a problem solver uses. In addition, any problem solver should be able to employ the computer during problem solving process according to her computation needs. That is, because all humans solve problems, all humans use the natural language of their problems domains and consequently all humans should be able to use their brain assistant, that is the computer, as a problem solving tool. Computational emancipation of problem domain, that is using the problem domain ontology, enables these desiderata because:

1. All concepts of problem domain can be stored in a file whose records are both concept terms and term meanings.
2. Computational emancipation mimics the human learning process, where knowledge are stored, both by terms and by meanings in a file, exactly like natural learning stores concepts by terms and by meaning in the brain.
3. Problem domain ontology allows us to disambiguate natural language and thus to develop software tools that make computer use as easy as the usage of any other tool developed by humans to strengthen them during problem solving activity.
4. Hence, computational emancipation is simple the process of natural learning using current computer technology.

This paper presents the methodology sketched in [3] towards this endeavor and therefore deserves to be largely known.

2 Introduction

The problem we address in this paper is the integration of the computer as a brain assistant within the human problem solving process. Original computers have not been developed as problem solving tools. Rather, computers were developed as number crunching tools to be used by mathematicians and engineers. The computer based problem solving methodology provided by the creators of the original computer consists of:

- Formulate the problem;

- Develop a solution algorithm;
- Encode the algorithm and its data into a *program* in the language of the computer;
- Let the computer execute the program;
- Decode the result and extract the solution of your problem.

This problem solving methodology offers computer programming as an “one-size-fits-all” pattern for computer use as a problem solving tool, independent of the problem domain. Therefore one may say that this paradigm of computer use *integrates the problem solving process within the computer*.

Successes of computer use during problem solving process have evolved software tools at the level of *information processing services* [4]. Moreover, currently the networking technology allows software tools to be exchanged as standalone pieces of composable tools called Web Services (WS). A new problem solving paradigm based on WSs emerges, where computer based problem solving process is split between problem domain expert and computer expert according to their expertise as follows:

- Domain expert formulates the problem and the solution algorithm in terms of problem domain concepts;
- Computer expert implements software tools and domain concepts as web services using computer languages;
- Computer user acquires and manipulates WSs in order to solve her problems.

The architecture of the problem solving software resulted depends upon the problem domain and evolves as a Service Oriented Architecture (SOA). The computer platform that runs it is transparent to the problem solver. Therefore, one may say that with this problem solving methodology *computer is integrated within the human problem solving process*. The problems raised by the interoperability of WS-s components of SOA-s are resolved using new standards. XML technology led to the development of three main standards that are used for the implementation of SOA-s:

1. Standard (Small) Object Access Protocol (SOAP), a standard that allows applications to invoke WS-s irrespective of the computer architecture on which they run.
2. Web Service Description Language (WSDL), a standard that allows software developers to describe WSs such that they can be discovered and used by other developers.
3. Universal Description, Discovery, and Integration (UDDI), a standard registry that allows software developers to advertise, sell, and buy WSs.

These standards transform computer based problem solving process into a computer business where the exchange unit is the WS. Unfortunately this computer business

is not targeted to the computer user. By the contrary, in addition to the language of the software tools, now computer user needs also to learn the intricacies of Web Programming, the language of the WSs and SOAs.

The recent hype about Cloud Computing (CC) promises to bring computers as problem solving tools to the masses. However, so far the main research on CC [5] concerns mostly cloud infrastructure management, expressed in terms of Virtual Machines (VMs) populating the cloud at a given time. But current VMs in the cloud context are abstractions of computer architectures not abstractions of problem domains. Therefore CC is addressed to computer experts not to problem domain experts. Moreover, the goal of CC is stated mostly in terms of computer resource optimization and efficiency, not in terms of how computer use can be addressed to masses. We believe that populating the cloud with domain dedicated virtual machines CC becomes a problem solving tool dedicated to masses.

3 Problem Solving Process

Problem and problem solving are among the few concepts computer scientists use without defining them, under the assumption that everybody understands them a priori. But for different domains of activity problem and problem solving may mean different things. For example, for a high-school student solving the equation $a * x^2 + b * x + c = 0$ means the development of the formula $x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4 * a * c}}{2 * a}$ which when fed with the coefficients a, b, c of the equation evaluates to the numbers x_1, x_2 that satisfy the equality $a * x^2 + b * x + c = 0$. On the other hand, for a computer expert this may mean the development of a program that inputs the numbers a, b, c and evaluates the expression $a * x^2 + b * x + c$ for all $x \in [Min_R, Max_R]$, where Min_R and Max_R are minimum and maximum real numbers representable in machine memory, and outputs those x for which the value of $a * x^2 + b * x + c$ is zero.

Teaching students the art of problem solving, Polya [6] has defined the concepts of a problem and problem solving as follows:

To have a problem means to search consciously for some action appropriate to attain a given aim. To solve a problem means to find such an action.

Notice that hidden here are three things: *unknown, action, purpose*. These concepts are independent of problem domain, therefore Polya's definition is robust. Polya's problem solving process involves the operations: identify the *unknown*, find the *action* that leads from the given *data* to the discovery of unknown, and check that the unknown thus found satisfies the *purpose*, i.e., satisfies the condition that characterizes the problem. Unknown, action, and purpose are natural language terms used to formulate and solve problems in any problem domain. In any scientific domain the natural language ambiguities in problem

formulation and solution algorithm development are resolved by the domain context. That is, for a mathematician an unknown may denote a mathematical abstraction while for a chemist it may denote a concrete chemical substance; the actions performed by the mathematician while developing a solution algorithm perform operations with mathematical abstractions while the actions performed by the chemist are operations with concrete physical instruments and chemical substances. Scientists solving problems manipulate the objects of their sciences whose meanings are different though their natural language notations may be the same. That is, though the natural language is infinite through the infinity of the discourse it manipulates, in any given domain the language used by the domain expert is unambiguous and is finitely generated by the mechanism of knowledge acquisition and use. Consequently, the problem solving process proposed by Polya is linguistically unambiguous and domain independent. Focusing on mathematical objects, Polya formulates it as the four steps problem solving methodology:

1. Formalize the problem;
2. Develop a plan (an algorithm) to solve the problem;
3. Perform the algorithm on the data characterizing the problem;
4. Validate the solution by checking the validity of problem conditions.

The requirement to formalize the problem means to express the three characteristic concepts components of the problem, unknown, condition, data, as mathematical objects. The result of “problem formalization” step depends upon mathematical knowledge and problem understanding. The requirement to develop a solution algorithm asks the problem solver to discover a sequence of well-coordinated operations which when applied to the data characterizing the problem leads to the values of the requested unknowns. The requirement to perform the algorithm asks the problem solver to actually execute the operations involved in the algorithm using her brain as a tool. This means to instantiate the problem by appropriate data, conditions, and unknown and to execute the operations defining the algorithm on the problem instance thus obtained. The requirement to validate the solution means to show that conditions characterizing the problem are satisfied by the solution discovered by the algorithm execution.

Computers evolved from tools that can help performing numerical operations to tools that can perform any kind of well-defined operation. Hence, computer can be used to help with algorithm execution irrespective of the problem and problem solving algorithm. To straighten the mechanism used by computers to perform operations during an algorithm execution, we give below an algebraic specification of a computer [7]:

```
beginSpec Computer  
name Hardware System is  
sort Memory, Processor, Devices, Control;
```

```

opns receive:Device × Control→ Memory;
        transmit:Memory × Control→ Device;
        store:Processor × Contro → Memory
        fetch:Memory× Control→ Processor;
        process:
            Memory × Processor × Control→ Processor,
            Memory × Processor × Control→ Memory
            Processor × Control→ Processor;
vars PC:Control;
axms PC.operation is receive or transmit or store or fetchi or process;
actn PEL:whilePluggedIn and PowerOn do
         $l_0$ : Perform(PC);PC:=Next(PC): $l_0$ 
endSpec Computer
    
```

The essential part is the action Program Execution Loop (PEL) composed of the functions **Perform()** and **Next()**. **Perform()** takes as the argument a control register called Program Counter (PC) and evaluates the operation encoded as its contents; **Next(PC)** determines the operation of the algorithm to be performed next. Computer Based Problem Solving Process (CBPSP) uses Polya methodology where problem solving algorithm is performed by a computer. This requires that problem characteristic components **unknown**, **data**, **condition**, as well as problem solving algorithm, be encoded in computer memory. The process of this encoding has been called the *computer programming*. In addition, a mechanism for activating the computer on a given program and for controlling computer's actions during program execution, must also be provided. This has been called the *program execution*.

Computer programming and program execution are tedious and error prone tasks, and they require problem solver to be a computer expert. So, to make computers usable by the humans, an evolving collection of programming tools have been developed as the *system software*. According to services provided for program development and execution, system software tools can be classified as *translators* and *operators*. Translators allow programmers to use high level mnemonic terms for machine operations during programming. Operators manipulate computer resources (memory, processor, devices, control, information) and events (interrupts and exceptions) that occur during program execution. But it doesn't matter the abstraction level of the terms used to denote computer resources, events, and system software tools, these terms represent computer elements and computer computation concepts. Software tools are not problem domain concepts. Therefore CBPSP actually embeds problem solving process into the computer language, irrespective of the problem it solves. To bring computers to masses it means to reverse this process, i.e., to embed the computer into the problem solving process. This is achievable by letting computer user employ the computer during the algorithm evaluation as a *brain assistant* that performs operations required

by the control flow of the algorithm evaluation. Current computer technology makes this task feasible by developing software tools that allow domain expert, computer expert, and computer network expert to share the problem solving process according to their domains of expertise, as follows:

- Domain expert formulates problems and develops solution algorithms using the problem domain logic;
- Computer expert develops software tools and provides them to computer users as web service;
- Computer network experts develop tools that allow problem solvers to ask computer networks to perform the tasks involved in their problem solving processes.

Irrespective of their expertise, all of them can use the computer as a brain assistant during their activity. The new software tools required by this computer based problem solving methodology are:

- The Domain Algorithmic Language (DAL), a computational language to be used by the problem solver to express problem solving algorithms.
- Computational Emancipation of the Application Domain (CEAD), which provides a data-representation of the problem domain that automates algorithm evaluation using a Domain Dedicated Virtual Machine (DDVM);
- The DAL System that implements the DDVM (in the cloud) and offers computer services to the computer user by subscription, without asking her to possess computer knowledge in order to consume these services.

In this paper we illustrate these tools using *the implementation of NLD System taking the Arithmetic as the problem domain*[8].

4 Domain Algorithmic Language

Polya's problem solving methodology is centered around problem formalization and problem solving algorithm development, using problem domain concepts. This is easily done for mathematical problems because mathematical well defined concepts are implicitly formalized. But for other problem domains, problem formalization and algorithm development may not be so obvious. However, whatever problem domain may be, problem formalization means define problem concepts and methods in terms of well-understood concepts and methods. Using a mathematical say, "one cannot expect to be able to solve a problem one does not understand". Our conjecture here is that solvable problems of any problem domain are expressible in terms of a finite number of well defined concepts. This is trivially true for the common sense problems raised by the usual real-life. A formal proof of this conjecture can actually be sought using decidability theory [9].

We assume further that for a problem solver, the problem domain consists of a set of well defined domain characteristic concepts, and is modeled by a tree as shown in Fig. 1.

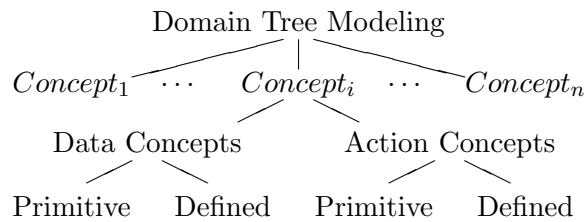


Fig. 1. Tree modeling of a problem domain

The *Primitive* leaves of the modeling tree represent domain characteristic concepts that are common to all domain experts. Primitive data are expressed by the concepts of variable and value. Primitive actions are expressed by the simple phrases of the form: $Subject \xrightarrow{Action} Object$, $Subject \xrightarrow{Property} Object$ where *Subject* and *Object* are data or actions (as appropriate), and \xrightarrow{Action} and $\xrightarrow{Property}$ are operations to perform or predicates to check, expressed by the common linguistic jargon of the domain. The *Defined* leaves of the modeling tree represent concepts created by problem solving and are specific to the problem solver. However, the mechanisms used to define new data and action concepts during problem solving are specific to the domain. We assume here that data definition mechanisms are formalized by mathematical concepts of *pair*, *vector*, *table*, *list*, *set*, *function*. Linguistic expressions of these definitions are domain characteristic, are tailored to the problem and, as appropriate, are formulated by the problem solver. The action definition mechanisms are formalized by mathematical rules that define the action-composition operations by *expression-well-formation*, *concatenation*, *choice*, *iteration*. The linguistic expressions of these definitions are domain specific phrases. Fig. 2 shows the example of tree modeling of arithmetic.

This domain modeling implies that the solution (algorithm) of any problem domain defines a new characteristic concept of that problem domain. Consequently, by problem solving, a problem domain becomes a potentially infinite collection of concepts usable to solve other potential problems of that domain. Problem solutions (algorithms) are expressed in terms of concepts and operations characteristic to the domain. These are actually valid expressions in the natural language of the problem solvers, which are understood by all domain experts because these expressions use only concepts familiar to the domain experts.

For example, for a high-school student learning arithmetic, the problem domain may be characterized by the set I of integer numbers with the operations $+ : I \times I \rightarrow I$, $- : I \times I \rightarrow I$, $* : I \times I \rightarrow I$. Then, solving the equation $a * x + b = 0$,

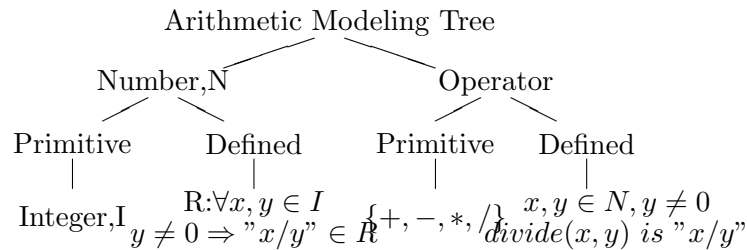


Fig. 2. Arithmetic modeling tree

$a, b \in I, a \neq 0$ means finding $c \in I$ such that $a * c + b = 0$. Using the properties of equality, the problem solver develops the formula $c = -b/a$. But one can easily observe that b/a is not always an integer. Therefore, problem solver concludes that $a * x + b = 0$ is not always solvable over the set of integers. However, if she extends I to R , the set of all real numbers, then the equation $a * x + b = 0$ is solvable and its solution is $x = -b/a$. Since division by zero is not defined, the problem solver requires the condition $a \neq 0$. Thus, by solving the problem a new well-defined concept, the set R of real numbers, has been developed and problem domain was enlarged with the new concept, Fig. 2.

The specification of the Domain Algorithmic Language (DAL) can be done using a vocabulary that contains language terms used for few characteristic concepts of the domain, and very simple rules for sentence formation. The potential ambiguity of these terms is eliminated by their meaning in the domain. In other words, though phrases containing these terms may be ambiguous as natural language expressions, these ambiguities are transparent for a domain expert. That is, for a problem domain D , $DAL(D)$ is the language spoken by an expert of the domain D .

The problem solving process expands the vocabulary of $DAL(D)$ with the terms used to name problem solutions. In addition, problem solution expressions (algorithms) expand the sentence formation rules with the rules provided by the solution expression. This mimics the natural learning process that characterizes the problem domain. We should observe here the difference between computer languages and DAL. Computer languages have a fixed vocabulary (lexicon) and a fixed set of algorithm well formation rules. DAL's vocabulary (lexicon) and the concept terms well formation rules evolves dynamically with the domain learning process.

Formally DAL may be specified using a pattern similar to the pattern used to specify computer languages, which consists of given a finite set of BNF rules specifying terms denoting domain characteristic concepts and few simple BNF rules for statement formation. Further, DAL specification mechanism allows both

its vocabulary and formation rules to grow dynamically with domain learning process. We call this the process of DAL's evolution. Since DAL terms and algorithms are natural language concepts (though they may have machine representations) domain experts can freely reuse them as components of the new concepts and solution algorithms developed during problem solving process, while preserving the unambiguity of DAL.

Grammatically, the initial terms of the DAL vocabulary would be categorized as nouns, verbs, adjectives, and adverbs. Here we choose the statement formation rules to fit the Resource Description Framework (RDF) used by the Semantic Web [10, 11], $Subject \xrightarrow{Action} Object$, $Subject \xrightarrow{Property} Object$, where *Subject*, *Object*, *Action*, *Property* are elements of the DAL vocabulary. But for any problem domain these rules can be chosen by domain expert collaborating with computer expert to fit the advances of their domain evolution. Of course, solution algorithms developed by the problem solving process are seen as statement formation rules expressed in terms of the already defined statement formation rules. The evolving DAL specification defined above could be best illustrated by any of the formal systems provided by the axiomatic specification of set theory [12].

Computational nature of DAL is obtained by DAL's semantics specification using a description logic [13] whose model is defined as follows:

- Implement every concept C of the DAL terminology as a web service $WS(C)$. Let $URI(C)$ be the URL of the $WS(C)$.
- Implement formation rules $Subject \xrightarrow{Action} Object$ by web services $WS(Action)$ whose input and output are elements of $Subject \times Object$.
- Implement formation rules $Subject \xrightarrow{Property} Object$ by web services $WS(Property)$ that input tuples of $Subject \times Object$ and return true or false.
- Implement every solution algorithm by a web service obtained by the composition of the web services employed in the algorithm using the following rules:
 1. Implement concept concatenation $C_1; C_2$ by service concatenation $WS(C_1); WS(C_2)$;
 2. Implement concept composition $C_1(C_2)$ by service composition $WS(C_1)(WS(C_2))$;
 3. For each domain specific operator, *rator*, implement concept composition $C_1 \text{ rator } C_2$ by a domain specific web service composition operator $WS(\text{rator})(WS(C_1), WS(C_2))$.

In order to allow algorithm evaluation by the problem solver using the computer as a brain assistant, we further structure DAL and its model using a domain ontology represented by a file in the Web Ontology Language, (OWL) [14]. For a problem domain D , let $OWL(D)$ be the OWL file representing the $DAL(D)$. A

solution algorithm in the domain D is then executed by the problem solver using an approach similar to the usage of a calculator to evaluate an expression. However, data and operations of the DAL algorithm are evaluated using computers available on the Internet and the OWL(D) as follows. Let \mathcal{A} be a solution algorithm to be executed.

1. Map the expression of \mathcal{A} into an expression tree. A Polish-form (prefix or postfix) can be used to express this tree. Let $PF(\mathcal{A})$ be the postfix form of the DAL algorithm.
2. Evaluate $PF(\mathcal{A})$ using a stack and OWL(D), by the following rules:
 - (a) Examine the $PF(\mathcal{A})$ from left to write.
 - (b) If a data concept d is examined, search d in the OWL(D). Let $URL(d)$ be the web service implementing the concept d . Call the web service at $URL(d)$ and push the result on the stack;
 - (c) If an action a (operation or property) is examined, search a in the OWL(D) and let $URL(a)$ be the web service implementing a . Call $URL(a)$ taking as input arguments the elements on top of the stack. Let r be the result. Delete the arguments taken as input by $URL(a)$ from the top of the stack and push r on the stack;
 - (d) The result of the DAL algorithm evaluation is on top-of the stack when the $PF(\mathcal{A})$ is completely examined.

This algorithm is well-known in compiler construction [15] and does not require any computer knowledge in order to perform it by a domain-expert. However, in this context the $PF(\mathcal{A})$ algorithm interpretation assumes that: (a) problem domain is represented as a data structure (the OWL file) that can be searched by the computer user, and (b) domain concepts are implemented as web services available on the Internet. Since computer user handles only domain concepts, this paradigm of computer use integrates the computer within the problem solving process.

Note: though the DAL algorithm evaluation described above follows a sequential approach, it can be implemented by a distributed system, as we shall see in Section 6.

5 Computational Emancipation of a Problem Domain

The DAL algorithm execution discussed in Section 4 demonstrates that current software technology allows computer integration within the problem solving process, as a brain assistant. But this integration lacks the efficiency because computer user spends all the time searching for web services in the OWL(D). In addition,

it imposes new complexities during problem solving determined by the structure of the OWL(D) and by the web service calling mechanism. Therefore, in order to be effective, this integration must be automated. How can this be done?

CEAD is the process that transforms the DAL from a fragment of natural language used by the problem solver into a computational language used to automate the problem solving process. Therefore CEAD can actually be seen as a new step towards domain formalization described in Section 4 and can be achieved by:

1. Software tools to automate the process of domain ontology creation and implementations;
2. Software tools that automate WS generation and optimize the search for the concept implementation in the domain ontology during the DAL algorithm execution;
3. Software tools that automate the process of WS evaluation during DAL algorithm execution;
4. Software tools that expand domain ontology with the terms denoting new algorithms developed during problem solving process and with the formation rules provided by these algorithms.

Many such software tools are already provided by current software technology. However, these tools have not been designed with this goal in mind. Therefore, while computer research creates tools dedicated to the goal set forth by the CEAD process, the challenge is to use the existing software as appropriate, in the context of the new problem solving methodology, which integrate the computer in the human problem solving process, further referred to as the Web Based Problem Solving Process (WBPSP).

5.1 Domain ontology

In this paper, domain ontology is a mechanism that facilitates the goal of domain algorithm execution, by the domain expert, employing the computer as a brain assistant, which uses web services to perform algorithm's operations. Therefore, while much of current work on ontology focuses on development and modeling [16, 17, 18, 19, 20, 21, 22] we concentrate on a domain ontology structuring and representation that supports the automation of concept identification in the domain ontology and the execution of the web services implementing domain concepts. Since WBPSP ensures domain evolution by the problem solving process, our ontology structuring must be automatically updated with the new concepts representing problems and solution algorithms. Hence, the ontology structuring we assume here is similar to that described in [23]. That is:

1. The domain ontology is specified by a taxonomy that is representable by a collection of disjoint trees whose nodes are primitive concepts of the domain and whose edges are relationships interpreted as logical subsumptions, that is to say that if concept C_1 subsumes concepts C_2 then $\forall x.C_1(x) \rightarrow C_2(x)$.

2. Ontology trees are of two kinds: DataConcept trees and ActionConcept trees. The relations among them are explicitly specified by their definitions. Examples of such definitions are the references to the input and the output of actions used in the domain algorithms.
3. New concepts are constructed by domain specific tree constructors which represent problem solving algorithms.

The methodology we use to build a domain ontology is similar to the “adaptive methodology” reported in [24] tailored to the goal of WBSP. That is, the domain ontology reflects the problem solving process which evolves the ontology by the user learning process, and thus consists of two parts: a part that represents the user own ontology and a part that represents the domain expert ontology. Domain Expert Ontology (DEO) is built by hand, using a small taxonomy chosen from a textbook, is evolved by the process of domain expert education, and is updated during problem solving process. This is performed by a collaboration between domain expert and computer expert as shown in Fig. 3.

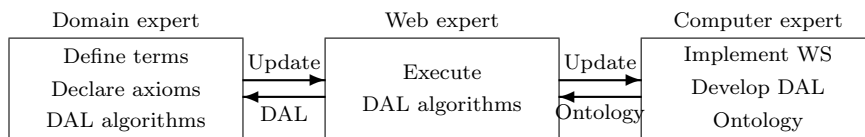


Fig. 3. Domain Ontology Implementation and Use

The User Own Ontology (UOO) is built automatically by tools from the DEO, thus extending automatically the TBox and the ABox during algorithm execution by the DDVMs. That is, initially UOO coincides with the DEO. Then, during problem solving process UOO is automatically expanded with new concepts representing problems and solution algorithms developed by the particular user. Hence, at a given time, the domain ontology consists of the core DEO, that is available to all domain users, and a private part (UOO) which is specific to a given domain user. The DEO may be extended by the system to represent the domain evolution containing the new domain discoveries developed by the activity of the collection of domain users. This may be illustrated with the evolution of arithmetic ontology to a vector space.

Consider an application domain modeled by a tree as shown in Fig. 1. Data concepts represent data that can be used in a computational process such as input and output of such a process. The three attributes of a data concept are: *type*, *value*, *literal*. The *type* of a data concept is defined by the collection of operations defined on that data; the value of the data concept is the abstraction it represents; and the *literal* is a string representing that data value during problem solving

process. For example, **Integer** type is defined by the collection of operations identified by +, -, *, / where +, -, * are total operations and / is a partial operation; **Integer** values are decimal numbers and are formally defined as *cardinals of sets*; **Integer** literals could be sequences of decimal digits (potentially prefixed by + or -) representing integer numbers. Using appropriate definitions, one can extend the primitive operations +, -, *, / to the operations *add, subtract, multiply, divide*, which are defined on **Number** that subsumes both **Integer** and **Rational**.

The CEAD process associates both data concepts and action concepts of the domain modeled with WS-s which represent their semantics. As suggested in Fig. 3, the WS-s are constructed by computer experts cooperating with domain experts. For example, the concepts in the arithmetic domain in Fig. 2 are modeled by WS-s automatically generated from Java classes as we shall see further.

Computationally domain ontology is a data base, such as a file. Further we use the OWL file to represent a domain ontology where concepts are represented by their properties. As seen above, a data concept such as **Integer**, has three main attributes: type, value, and literal. These attributes are represented in OWL language by three properties: **hasType**, **hasValue**, and **hasLiteral**. Here we use the RDF triples to represent these properties which look like: $\text{Integer} \xrightarrow{\text{hasType}} \text{URI}(\text{integerType})$, $\text{Integer} \xrightarrow{\text{hasValue}} \text{URI}(\text{integerValue})$, and $\text{Integer} \xrightarrow{\text{hasLiteral}} \text{URI}(\text{integerLiteral})$. The action concepts like **add**, **subtract**, **multiply**, etc., are associated with WS-s which implements them via a Concept Agent. There could be several WS instances that implement the same concept so that if one instance is down other instances can take over. For example, the concept **add** may have the agent **addAgent** implemented by two WS instances: **addInstance1**, **addInstance2**. The agent maintains the list of web services which it can execute as implementations of the action it performs. Therefore, the RDF triples that define an action concept **a** in the OWL file will look like: $\mathbf{a} \xrightarrow{\text{hasAgent}} \mathbf{aAgent}$ and $\mathbf{aAgent} \xrightarrow{\text{implementedBy}} \mathbf{aInstance}_1; \dots; \mathbf{aAgent} \xrightarrow{\text{implementedBy}} \mathbf{aInstance}_n$. For example, the **add** action of the **Integer** type is represented in OWL by the RDF triples: $\mathbf{add} \xrightarrow{\text{hasAgent}} \mathbf{addAgent}$, and $\mathbf{addAgent} \xrightarrow{\text{implementedBy}} \mathbf{addInstance}_1, \mathbf{addInstance}_2$. The signature $\text{Integer} \times \text{Integer} \xrightarrow{\text{add}} \text{Integer}$ of the **add** action is represented in the OWL file using the three RDF triples: $\mathbf{add} \xrightarrow{\text{hasInput}} \mathbf{IntegerPair}$, $\mathbf{add} \xrightarrow{\text{hasOutput}} \mathbf{Integer}$, and $\mathbf{add} \xrightarrow{\text{hasAgent}} \mathbf{addAgent}$.

5.2 Using protégé for OWL file development

There are no software tools created to automatically generate an OWL file. Therefore we use Protégé to create and update an OWL file representing a domain ontology. Hence, an OWL file representing a domain ontology is composed of a header and a body. The header tells us about the namespaces used in the ontology

document and the ontology documents imported in the ontology document. Each namespace is specified by a **Prefix** construct and each ontology imported is specified by an **Import** construct. The body is basically composed of entity declarations (classes, properties, objects, individuals, axioms). Such declarations are in the form of RDF triples. We may use either XML syntax or OWL 2 Manchester Syntax [25] to express them. Though XML syntax is verbose, we believe that it is better understood by people and therefore we use XML syntax in the examples that follow. Since the goal of this paper is to describe a system that allows a computer user to perform problem solving using her computer as a brain assistant, we simplify the concept representation in OWL language and split the activity of OWL file creation in two steps. The first step is where the domain concepts are represented in the OWL file without being associated with web services implementing them, and the second step is where concepts in the OWL file are associated with their semantics. The first step is automatically performed by domain expert using Protégé tool [26], and second step is performed by the computer expert collaborating with domain expert. So far there are no tools assisting this activity. However, as we shall see in the next section, such tools can be easily developed.

Protege is an ontology editor tool which provides Graphical User Interface (GUI) so that the process of editing OWL files is easier. The user can create the OWL file by entering each concept as a class via the Protégé GUI. The subsumptions relation present in the domain model is called the sub-class relation in Protégé. The major benefit of using Protégé for the first step of the OWL file development is automatic creation of the OWL ontology file header. An example of an OWL file as created by Protégé is shown below. To gain space we collected all constructs **Class:concept** on one line though Protégé would place each of them on its own line.

```
Prefix: xml: <http://www.w3.org/XML/1998/namespace>
Prefix: xsd: <http://www.w3.org/2001/XMLSchema#>
Prefix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
Prefix: rdfs: <http://www.w3.org/2000/01/rdf-schema#>
Prefix: owl: <http://www.w3.org/2002/07/owl#>
Prefix: <http://bula1.cs.uiowa.edu/ontologies/arithmetics.owl#>

Ontology: <http://bula1.cs.uiowa.edu/ontologies/arithmetics.owl>
Class:Integer Class:add Class:multiply Class:subtract Class:divide
```

5.3 Updating an OWL file with web services

The file created by Protégé in the first step of the CEAD process represents a *pure* domain ontology where concepts are predefined (primitive) and are not associated with their implementations. We denote this file by `domainPURE.owl`. The second

step of the CEAD process consists of creating the file `domainCEAD.owl`. This is initiated by including in the `domainCEAD.owl` the file `domainPURE.owl`. Then the entities in the file `domainCEAD.owl` are associated with computer artifacts implementing them, thus performing the second step of the CEAD process. This activity is standardized by the two kinds of knowledges we are handling: data concepts and action concepts. The patterns used to specify data concepts and action concepts consist of sequences of text lines where:

1. First line represents the domain term used to denote the concept;
2. Each line that follows represents a property (in the sense of OWL) of the concept specified on its previous lines. We use indentation conventions for the identification of the domain and range of the property, as follows:

```
Property term
  Property Domain
  Property Range
```

Since we use WS-s as semantics of data concepts the primitive data are supplied by XML schema and are: `xsd:int`, `xsd:double`, `xsd:boolean`, `xsd:string`, `xsd:time`, etc., (see XML schema). All the other concepts are represented in terms of the predefined or already defined concepts.

The two kind of patterns that represent the two kind of concepts are:

1. Data concepts are specified by the pattern:

```
DataConcept: domain term used to denote it
  Individuals: concept term itself as an individual
  Objects: instantiations of the DataConcept
    Individuals: object's elements
    Constant: an immutable element
    Variable: a mutable element
  Type: Set of action concepts available on the objects
```

Example data concept is the Integer which when fit in the above pattern becomes:

```
<owl:Class rdf:ID="Concept">
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#DataConcept" />
    <owl:Class rdf:about="#ActionConcept" />
  </owl:unionOf>
</owl:Class>
<owl:Class rdf:ID="ActionConcept"/>
<owl:Class rdf:ID="DataConcept"/>
<owl:Class rdf:ID="Input"/>
```


2. Properties of the concepts are defined as XML elements whose tags are OWL properties `ObjectProperty`, `DataProperty`, `FunctionProperty`, whose ID attribute identifies the property name, such as `hasInput`, `hasOutput`, etc., and the XML element components define the domain, range, and the type of the property. Example property definitions are:

```
<owl:ObjectProperty rdf:ID="hasInput">
  <rdfs:domain rdf:resource="#ActionConcept"/>
  <rdfs:range rdf:resource="#Input"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="inputName">
  <rdfs:domain rdf:resource="#Input"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
<owl:FunctionalProperty rdf:ID="serviceName">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DataProperty"/>
  <rdfs:domain rdf:resource="#ServiceInstance"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:FunctionalProperty>
```

The description of namespaces, concepts, and properties in `cead.owl` follows a standard pattern. Therefore the examples given above are sufficient to understand the concept representation in `arithmeticCEAD.owl` that illustrates the `domainCEAD.owl` file. Here we illustrate WS generation for domain's primitive concepts using the file `ArithmeticsPure.owl`. To simplify the matter we show only the web services associated with the data concept `Integer` and action concept `add`, and use XML syntax which we believe is more accessible to the reader. The rest of the entities of the `ArithmeticsPure.owl` ontology are treated similarly.

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns="http://bula1.cs.uiowa.edu/ontologies/arithmetic.owl#"
  xmlns:sadl="http://bula1.cs.uiowa.edu/site/sadl.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:xs="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://bula1.cs.uiowa.edu/ontologies/arithmetic.owl"
  xmlns:tns1="http://webservices.nld.cs.uiowa.edu/">

  <sadl:DataConcept rdf:about="#Integer">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
    <sadl:description> Integer concept of arithmetics </sadl:description>
    <sadl:dataType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      xs:int
    </sadl:dataType>
  </sadl:DataConcept>

  <sadl:ComputationalConcept rdf:about="#Add">
    <sadl:description> Returns the sum of two integers </sadl:description>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
```

```

    <sadl:hasInput>
      <sadl:Input rdf:ID = "addI1">
        <sadl:inputType rdf:resource="#Integer"/>
        <sadl:order> 1 </sadl:order>
      </sadl:Input>
    </sadl:hasInput>
    <sadl:hasInput>
      <sadl:Input rdf:ID = "addI2">
        <sadl:inputType rdf:resource="#Integer"/>
        <sadl:order> 2 </sadl:order>
      </sadl:Input>
    </sadl:hasInput>
    <sadl:hasOutput rdf:resource="#Integer"/>
    <sadl:hasProfile>
      <owls2:Profile rdf:ID="addProfile">
        <cead:implementedBy rdf:resource="#addServiceInstance1"/>
      </owls2:Profile>
    </sadl:hasProfile>
  </sadl:ComputationalConcept>
  <sadl:ServiceInstance rdf:ID="addServiceInstance1">
    <sadl:uri rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      http://localhost:8080/ArithmeticsWebServices/calculator
    </sadl:uri>
    <sadl:wSDLFile rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      http://localhost:8080/ArithmeticsWebServices/calculator?wsdl
    </sadl:wSDLFile>
    <sadl:serviceName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      Calculator Service
    </sadl:serviceName>
    <sadl:portName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      CalculatorServiceHttpSoap12Endpoint
    </sadl:portName>
  </sadl:ServiceInstance>
  ...
  Copy all operatios
Copy concepts
</rdf:RDF>

```

For convenience, the domain ontology file is associated with two dedicated namespaces: a names space called vocabulary, where all basic terms of the domain are collected, and a name space where the URI-s of the WS-s implementing the terms in teh vocabulary are collected. Since our example regards arithmetic domain and the WS-s implementing its terms are performed on the computer named *bulai1* at the site *cs.uiowa.edu* using the *sadl* virtual machine, described in section 6, the terms we used for these spaces are: <http://bula1.cs.uiowa.edu/site/arithmeticCEAD.owl> and <http://bula1.cs.uiowa.edu/site/sadl.owl>.

The computer artifacts used to represent concept semantics in the domain ontology file are in general developed by computer experts collaborating with domain experts. They can use any tools to implement them. The computer technology abounds of such tools [27, 28, 29, 30] and many other. These tools allow computer experts to develop WS by hand or to automatically generate them from conventional computer artifacts such as programs written in various programming languages as are Java, C, C++, etc. Among these tools Apache Axis is a light-weight, yet powerful tool for automatic WS generation from plain Java classes or C functions. We develop WS-s used to CEAD a domain of interest using two approaches:

1. WS-s associated with primitive concepts are automatically generated from Java programs using Apache Axis technology.
2. WS-s associated with user concepts defined during problem solving process are automatically developed by our own method as we shall see in section 6.

6 Domain Dedicated Virtual Machine

The efficiency of the DAL algorithm execution by problem solver using the computer as a brain assistant is improved by associating each concept used in the $PF(\mathcal{A})$ with the WS that implements it. This can be easily done by hand, by the problem solver, or by an appropriate automaton that operates on $PF(\mathcal{A})$ and $OWL(D)$. The result can be seen as a “program” in the language of the brain assistant used by problem solver to execute the DAL algorithm. However, since the operations performed by this automaton (the brain assistant) are WS-s implementing the concepts of the problem domain, we call it the Domain Dedicated Virtual Machine (DDVM).

Formally, DDVM can be seen as a tuple $DDVM = \langle ConceptC, Execute, Next \rangle$ where:

- `ConceptC` is a Concept Counter, that, for a given DAL algorithm \mathcal{A} , points to the web service in the $OWL(D)$, that implements the concept, to be performed next during the algorithm execution;
- `Execute()` is the process that execute the computations in the WS pointed to by `ConceptC`;
- `Next()` is a function which determines the next concept of the DAL algorithm \mathcal{A} to be performed by `Execute()` during algorithm execution.

The DDVM performs similarly with the PEL (see Section 3) and therefore the algorithm execution by DDVM can be described by the following Domain Algorithm Execution Loop (DAEL):

```
ConceptC = FirstDALConcept(DAL algorithm)
while (ConceptC is not End)
    Execute (ConceptC);
```

```
ConceptC = Next(ConceptC, DAL algorithm)
Extract result + display it to the user
```

On closer inspection one can easily see the similarity between DDVM and a Virtual Monitor [31]. The ConceptC is an abstraction of the program counter, the WS pointed to by the ConceptC is similar to the function executed by the OS simulating instructions of the machine implemented by the VM, and Next() is similar to the process that determines the next instruction of the program run by the VM. The difference is that the memory of the machine implemented by DDVM is the OWL(D), the processor of the DDVM is the collection of all processors available on the Internet (in the cloud) that implement WS-s used in the OWL(D), and the Next() is well defined by the relationship of the data and operations in the Polish form of the DAL algorithm expression. Therefore, the DDVM is a true domain dedicated virtual machine.

Once an application domain is CEAD-ed, the automation of DAL algorithm execution is based on two main software components:

1. a translator that maps the DAL algorithm into an expression tree whose nodes are labeled by domain concepts associated with the URL of the WS-s implementing them, and
2. an interpreter operating on the expression tree generated by the translator, executing WS-s encountered at the tree nodes.

The translator is implemented by the conventional compiler construction tools and the interpreter is implemented by a stack machine similar to Java Virtual Machine (JVM).

For a given DAL algorithm \mathcal{A} the mapping of \mathcal{A} into the expression tree $ET(\mathcal{A})$ is automatically performed by the DAL parser, that transforms \mathcal{A} into its parse tree, $PT(\mathcal{A})$. A bottom-up traversal of the $PT(\mathcal{A})$, that searches the OWL(D) for the domain concepts used in the $PT(\mathcal{A})$ and associates them with the URL of the WS-s implementing them, maps the parse tree $PT(\mathcal{A})$ into the expression tree $ET(\mathcal{A})$. The automation of the DAL algorithm execution using the WS-s available on the Internet requires the $ET(\mathcal{A})$ to be transformed into an appropriate language that has WS-s as operations performed by DDVM. For this purpose we use the Software Architecture Description Language (SADL) [32, 33].

6.1 Software architecture description language

Software Architecture Description Language (SADL), inspired by Armani [34], has been conceived as a language suitable to describe functional behavior of component-based software architectures, where components are standalone and composable pieces of software. Hence, its goal is similar to the goal of the Intermediate Language (IL) used by Microsoft's ASP.NET Framework. However,

SADL evolved as a language suitable to describe functional behavior of component-based software architectures, where components are Web Services. Consequently the SADL software is designed to run on the network, therefore compiler construction technology provides a suitable mechanisms to implement it.

As any language, SADL syntax has a three layer structure: vocabulary, simple constructs, and composed constructs. SADL vocabulary is a dynamic collection of terms used to denote problem domain concepts. Since SADL is meant as the target for any DAL implementation, it needs to be implemented as a domain dedicated namespace where each terms is associated with the collection of semantic properties that defines it in the respective domain. For example the term `Integer` in the SADL namespace of the High-School Arithmetic is specified by:

```
<cead:DataConcept rdf:about="#Integer">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <cead:description>
    This is the integer number concept in arithmetics domain.
  </cead:description>
  <cead:type rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    xs:int
  </cead:type>
</cead:DataConcept>
```

SADL vocabulary is the collection of DAL terms used by problem solvers in their DAL algorithms during problem solving process. Thus, from a computational viewpoint SADL terms denote computer process names. The code executed by these processes is associated with the term in the SADL namespaces and specifies completely the WS implementing that term. For example, the process executing the integer addition is associated with the term `addI` as follows:

```
<cead:ActionConcept rdf:about="#addI">
  <cead:description>
    This is the add operation in the arithmetics domain.
  </cead:description>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <cead:hasInput rdf:ID = "orderedPair">
    <cead:pairFirst rdf:resource = "#Integer"/>
    <cead:pairSecond rdf:resource = "#Integer"/>
  </cead:hasInput?>
  <cead:hasOutput rdf:resource = "#Integer"/>
  <cead:hasAgent>
    <cead:Agent rdf:ID = "addAgent"/>
    <cead:ImplementedBy = rdf:resource = "#addServiceInstance1"/>
  </cead:hasAgent>
</cead:ActionConcept>
<cead:ServiceInstance rdf:ID="addServiceInstance1">
  <cead:wSDLFile rdf:datatype = "http://www.w3.org/2001/XMLSchema#string">
```

```
    http://bula1.cs.uiowa.edu:8080/axis2/services/CalculatorService?wsdl
  </cead:wSDLFile>
  <cead:serviceName rdf:datatype = "http://www.w3.org/2001/XMLSchema#string">
    CalculatorService
  </cead:serviceName>
  <cead:operationName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    add
  </cead:operationName>
  <cead:portName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    CalculatorServiceHttpSoap12Endpoint
  </cead:portName>
</cead:ServiceInstance>
```

The simple constructs of the SADL are simple XML elements: `<tag attributes />` where `tag` is a term in the SADL namespace and each attribute is a tuples of the form `property = "value"` where `property` is a property of the process (data are considered here as unary operations) represented by the term `tag`. For example, the process that perform the addition of two integers is specified by: `<ari:addI input = "x, y" output = "z"/>` where `ari` is the prefix of the arithmetic vocabulary namespace.

The composed constructs of the SADL language are XML constructs composed with the terms: `foreach`, `if`, `ifthen`, `next`, etc. Example, the SADL expression of the formula: $x1 = (-b - \sqrt{b^2 - 4 * a * c}) / (2 * a)$ is represented by the following XML code:

```
<ari:delta input="a, b, c" output="delta" />
<ari:sqrt input="delta" output="tmp1" />
<ari:unaryMinus input="b" output="tmp2" />
<ari:subtract input="tmp2, tmp1" output="tmp3" />
<ari:multiply input="2, a" output="tmp4" />
<ari:divide input="tmp3, tmp4" output="x1" />
```

Note that SADL composition operators are provided as tags in the SADL namespace, as any other term of the problem domain.

SADL expressions are SADL representations of DAL algorithms.

6.2 SADL interpreter

SADL interpreter inputs a SADL expression and interprets it on a stack, in a manner similar to the byte-code interpretation of a Java code. Since each SADL simple element composing a SADL expression represent a process executed on interned, the flow of control during a SADL expression evaluation requires synchronization of these processes. Thus, the SADL interpreter performs a distributed implementation of the DAL algorithm. The simplest synchronization mechanism

used to control the flow of processes performing a DAL algorithm is provided by Unix `wait` and `, signal` primitives inserted in the SADL expression, after each SADL simple element. While this SADL implementation performs DAL algorithm distributed, on Internet, the algorithm execution is restricted to being sequential, where the computation unit is the WS. This mechanism can be extended to allow the processes performing a DAL algorithm to perform in parallel.

6.3 Evolving domain ontology

One of the key ideas of the DAL system is to provide a method that allows domain experts to create and extend their own CEAD-ed domain knowledge base. The DAL system solves this problem by allowing domain experts to create new action and data concepts.

6.3.1 Creating new action concepts

In order to create a new action concept, first of all, a domain expert expresses the new concept by an DAL expressions which is then saved in a file. Then she adds the concept to her UOO via an DAL Console program by executing "add2Onto <file>" command. This command translates the DAL expression into a SADL expression and sends it to her private space in the cloud, to which she subscribed. An Ontology Manager in the cloud automatically analyzes the submitted SADL expression and creates a new domain concept in the user's UOO with a name specified in the DAL expression. The Ontology Manager also creates a web service broker which wraps around the SADL code so that the concept is available on the Internet as a standalone, composeable software component. All the information about this concept's web service is automatically linked back to the user's UOO so that newly created domain concept is CEAD-ed. From now on, the user can use that new concept as any other CEAD-ed domain concepts such as using it in a DAL Consoles or composing it with other CEAD-ed domain concepts in an DAL expression to express the user's new computation.

The above scenario is demonstrated with the example in high school algebra that maps the algorithm solving quadratic equations into a new concept called `Solver`. We assume that the DAL expression of the algorithm that solves quadratic equations is written as follows and saved as the file `solver.nld`:

```
algorithm "Solver";
description: "This is a quadratic equation solver.";
message: "Provide coeffs of equation ax^2 + bx + c = 0";
input: a, b, c real;
output: x1, x2 real;
local: t real;
t = b * b - 4 * a * c;
if t >= 0 then
```

```
x1 = (-b - sqrt(t)) / (2 * a);
x2 = (-b + sqrt(t)) / (2 * a);
else
  print "the equation has no real solution";
endif;
```

Then using the DAL Console the user executes the command

```
add2Onto solver.nld
```

With the help of user's profile, including user's CEAD-ed ontologies and dictionaries, the DAL Console translates the above DAL expression into the following SADL expression:

```
<?xml version="1.0" encoding="UTF-8"?>
<sadl xmlns:xs="http://www.w3.org/2001/XMLSchema" name="Solver">
  <imports>
    <import type="ontology" uri="http://bula1.cs.uiowa.edu/owl/arithmeticPURE.owl"/>
    <import type="ontology" uri="http://bula1.cs.uiowa.edu/owl/arithmeticCEAD.owl"/>
    <import type="ontology" uri="http://bula1.cs.uiowa.edu:8080/NLDPortal/profile/<user>/PURE.owl"/>
    <import type="ontology" uri="http://bula1.cs.uiowa.edu:8080/NLDPortal/profile/<user>/CEAD.owl"/>
  </imports>
  <declaration>
    <inConst a, b, c type = "xs:double" />
    <outVar x1, x2 type = "xs:double" />
    <localVar t0 type = "xs:boolean" />
    <localVar t1, t2, t3, t4, t5 type="xs:double" />
  </declaration>
  <perform>
    <ari:delta input="a, b, c" output="t1" />
    <ari:greaterOrEqual input="t1, 0" output="t0"/>
    <ifTrue "t0">
      <perform>
        <ari:unaryMinus input="b" output="t2" />
        <ari:sqrt input="t1" output="t3" />
        <ari:add input="t2, t3" output="t4" />
        <ari:subtract input "t2, t3" output = "t5" />
        <ari:multiply input="2, a" output="t2" />
        <ari:divide input="t4, t2" output="x1" />
        <ari:divide input="t5, t2" output="x2" />
      </perform>
    <else>
      <perform>
        <print message="the equation has no real solutions" />
      </perform>
    </else>
  </if>
</perform>
</sadl>
```

This SADL expression is then sent to the Ontology Manager in the cloud. The Ontology Manager analyzes the SADL expression and create a web service broker for this SADL expression at the URL address

```
http://bula1.cs.uiowa.edu:8080/NLDPortal/profiles/<user>/services/Service?wsdl
```

The Ontology Manager also creates a new entry in the user private ontology (<user>PURE.owl and <user>CEAD.owl) as follows:


```
<cead:ActionConcept rdf:about="#Solver">
  <cead:description>
    This is a quadratic equation solver.
  </cead:description>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <cead:hasInput rdf:List = "a, b, c" rdf:resource = "#Real"/>
  <cead:hasOutput rdf:List = "x1, x2" rdf:resource= "#Real"/>
  <cead:hasAgent>
    <cead:Agent rdf:ID="solverAgent">
      <cead:implementedBy rdf:resource="#solverServiceInstance1"/>
    </cead:Agent>
  </cead:hasAgent>
</cead:ActionConcept>
<cead:ServiceInstance rdf:ID="solverServiceInstance1">
  <cead:wSDLFile rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    http://bulal.cs.uiowa.edu:8080/NLDPortal/profiles/<user>/services/Service?wsdl
  </cead:wSDLFile>
  <cead:serviceName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Solver
  </cead:serviceName>
  <cead:operationName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    solver
  </cead:operationName>
  <cead:portName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    ServiceHttpSoap12Endpoint
  </cead:portName>
</cead:ServiceInstance>
```

Now the user can use the concept "Solver" as any other primitive concepts by executing the command `use Solver`. The user can also use this concept in another DAL expression as shown by the following example:

```
x = Solver(a, b, c);
print "First solution of the equation: ";
print x.x1;
print "Second solution of the equation: ";
print x.x2;
```

6.3.2 Creating new data concepts

DAL System is also provided with the mechanism that allows a user to add data concepts to her UOO. New data concepts must be defined as compositions of other known data concepts using such definition schemes as **record**, **vector**, **set**. Since all the known data concepts are represented as some XML data type, the DAL system represents the new data concept using an appropriate constructor **record**, **vector**, **set** that maps the user defined data concept into an XML data type. The method for a user to create a new data concepts are described in the following steps:

1. The user defines the new concept in an DAL expression as shown in the above pattern.
2. The user use a DAL Console to submit the DAL expression to her private space in the cloud.
3. The Ontology Manager in the cloud receives the DAL expression and analyze it.
4. When the Ontology Manager finds a data concept definition:
 - (a) creates the corresponding domain data concept and add to the user's UOO.
 - (b) creates a new XML Data type which represents the data concept following the above pattern.
 - (c) automatically link the newly created data concept with the corresponding XML Data type.
5. The CEAD-ing process for creating new data concept finished.

We illustrate the mechanism of extending domain ontology with new data concepts with the example where a user defines the data concept **Complex** that represents complex numbers in the high school arithmetic domain. Since a complex number is a record of two real numbers the user defines the concept **Complex** using the following DAL expression:

```
concept Complex is
  record
    ImgPart integer;
    RealPart integer;
  endrecord;
endconcept,
```

The XML schema used to transform this DAL expression into a SADL expression is:

```
<xs:schema attributeFormDefault="qualified"
  elementFormDefault="qualified" targetNamespace="some-URI">
  <xs:complexType name="NewDataConceptName">
    <xs:sequence>
      <xs:element minOccurs="0" name="fieldName1" type="fieldType1"/>
      <xs:element minOccurs="0" name="fieldName2" type="fieldType2"/>
      ...
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

In the case of the **Complex** concept, we have the following concrete XML data type definition:

```
<xs:schema attributeFormDefault="qualified"
            elementFormDefault="qualified"
            targetNamespace="some-URI">
<xs:complexType name="Complex">
  <xs:sequence>
    <xs:element minOccurs="0" name="imgPart" type="xs:double"/>
    <xs:element minOccurs="0" name="realPart" type="xs:double"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>
```

7 DAL System

DAL System provides a user-dedicated implementation of a computer. That is, a computer user who install this system on her computer can further use the computer as a brain-assistant dedicated to her problem domain. Since the computer use lacks the efficiency when used in this manner we chose to describe here the implementation of the system in the cloud. This manner of DAL System implementation dedicates the system to a problem domain, thus allowing the computer to be shared among many users, who in effect share the problem domain in a manner in which the students of a class share the class instructor's knowledge.

7.1 Cloud implementation of DAL system

Cloud-implementation of the DAL System is described in Fig. 4. The assumption is that CC that accommodates the DAL System would have an administrator that manage the system allowing various users to register for DAL System use on a given problem domain. For that the CC is provided with a data base where all the SEAD-ed domain ontologies are maintained. The user subscription for a domain D is performed by an installation procedure that activates DAL System with the domain ontology required.

Further, as shown in Fig. 4, the user customizes the system to her personal use, evolving the problem domain she subscribed for with the concepts she learned and/or created during her own problem solving process. When the user decides to leave the system and cancel her subscription, the DAL System's manager may buy the knowledge developed by the user and update the domain, thus ensuring domain evolution with the concepts developed by the respective user. This ensures a domain evolution with the knowledge developed by problem solving process of all domain experts.

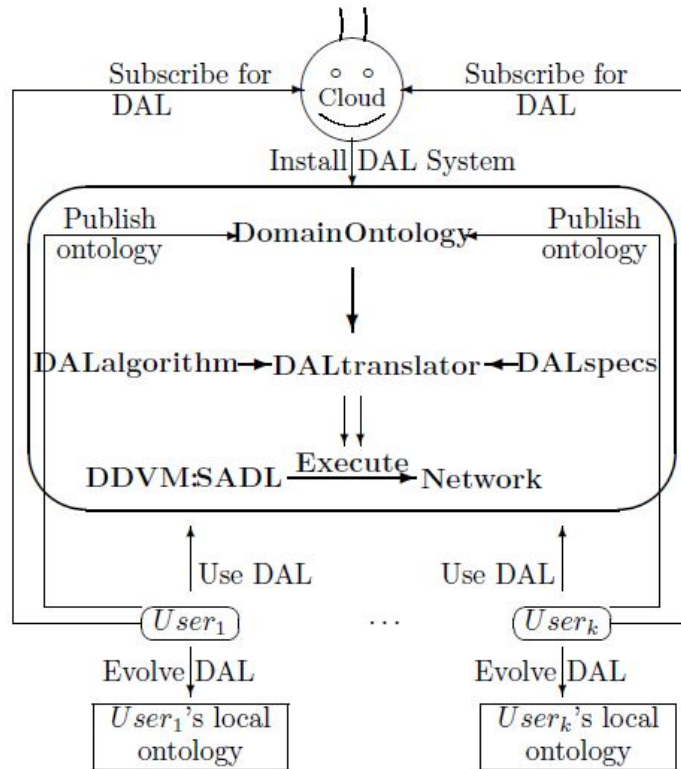


Fig. 4. Architecture of an DAL System

7.2 User interaction with DAL system

A user doesn't need a computer in order to interact with the DAL System. An iPad (or any other display) which provide a two-way communication using a command language can be used in this purpose. We envision here a Unix shell interaction as described in Fig. 5.

The DAL System is not appropriate for iconic-language implementation because it manipulates concepts that can be created by the user. Since the system is natural language based, and natural language is infinite through the infinite sequences of human generations speaking it, Window-implementation, though possible, would not be appropriate.

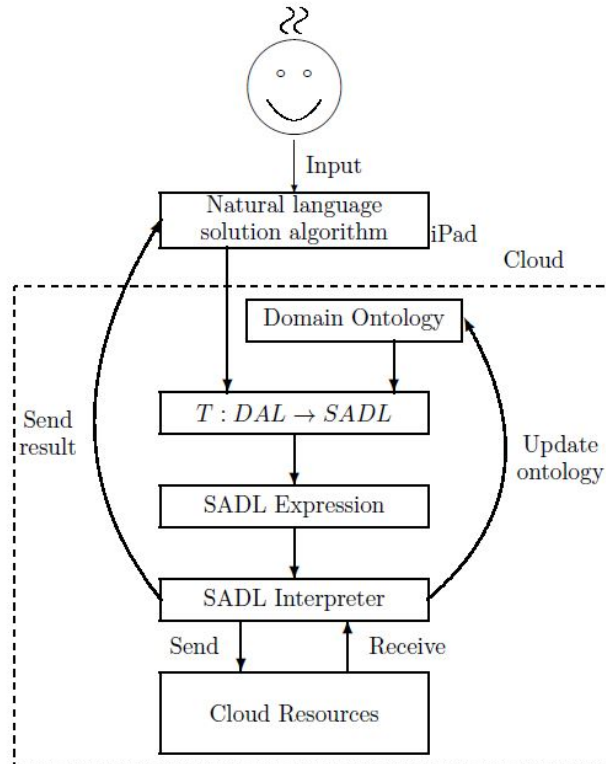


Fig. 5. Interacting with an DAL System

8 Conclusions

The computer, as the automobile, is a human created tool used to help people solve their problems. But unlike the automobile, the problem where computer is used is not specified by the term *computer*. Therefore while the automobile is integrated in the process of problem solving by the natural language term *automobile*, the computer creators have developed a special methodology for its use which is not integrated in the process of problem solving. By the contrary, the methodology developed by computer creators for its usage integrates the computer based problem solving process within the computer language. This implies that computer use for problem solving requires knowledge of computer architecture and functionality. This is acquired by high-level education which implies that computer can be used only by the people specially educated in this purpose. But all people need to solve problems, so current methodology for

computer use is discriminatory. In addition, complexity of software tools created by computer scientists to make computer use easier, increases with the complexity of the problem domains of computer uses thus strengthening computer technology itself. The research reported in this paper has the aim to discuss an alternate methodology for computer use by the integration of the computer within the problem solving process, not vice-versa, thus using the natural language of the user. This is achieved by the following steps:

1. Develop the Domain Algorithmic Language (DAL), which is an unambiguous sublanguage of the natural language of the problem solver. The syntax of this language is mathematically specified by an unambiguous grammar, and its semantics are specified based on research on the Semantic Web.
2. Develop an ontology where the concepts of DAL are stored as tuples $\langle \text{Term}, \text{Meaning} \rangle$, where the Term is the natural language term of the concept, and the Meaning is a Web Service implementing the concept. The process of DAL representation by such an ontology is called Computational Emancipation of the Application Domain (CEAD). The CEAD-ing process is performed by the domain problem solver during her school education, under the direction of domain experts as teachers, and continues during problem-solving. This allows DAL to be dynamically evolved and customized to the domain user. The conclusion is that the education of the domain problem solver is initiated in school and continues as long as that problem solver uses the domain as her profession.
3. In addition to the CEAD, the domain is provided with a Domain-Dedicated Virtual Machine (DDVM), which is a virtual computer whose memory is the domain ontology, and whose instructions are the terms of the domains and their execution is performed by the execution of the Web Service associated with the term.
4. The problem-solving process consists of using DAL to express the solution to her problem. Since DAL is a natural language, this process ends up with a DAL algorithm. Furthermore, the DAL algorithm is executed using real computers on the computer network by the computation involved in the Web Services populating it.

The conclusion is that the computer user uses only her natural language while solving problems with her computer. This means that the computer is integrated within the problem-solving process performed during DAL algorithm execution. The feasibility of this problem-solving methodology is discussed in the paper by showing the required software tools and by presenting a system that implements this methodology for the domain of problem solving in high-school algebra.

Competing Interests

Author has declared that no competing interests exist.

References

- [1] Rus T. Computer integration within problem solving process. In Proceedings of RoEduNet 11-th International Conference 2013 Jan 6.
- [2] Falbo RA, Quirino GK, Nardi JC, Barcellos MP, Guizzardi G, Guarino N, Longo A, Livieri B. An ontology pattern language for service modeling. In Proceedings of the 31st Annual ACM Symposium on Applied Computing 2016; 321-326. Available: <https://doi.org/10.1145/i2851613.2851840>
- [3] Rus T. Milestones for Computing Future.' Open Journal of Software Engineering and Applications. 2016;9:52—56. Available: <https://doi.org/10.4236/jsea.20169200>.
- [4] Software as a Service (SAAS). Available: en.wikipedia.org/wiki/Software_as_a_service, 2010.
- [5] Srinivasan S, Getov V. Navigating the cloud computing landscape: technologies, services, and adopters. Computer. 2011;44(3):22-3.
- [6] Polya G. How To Solve It. Second Edition. Princeton University Press, second ed., Princeton; 1973.
- [7] Rus T, Rus D. Systems Methodology for Software. World Scientific, Singapore; 19930.
- [8] Bui CK. An Evolutional Domain Oriented Approach to Problem Solving Based on Web Service Composition. PhD Thesis, The University of Iowa, Department of Computer Science, Iowa City; 2013.
- [9] Sipser M. Introduction to the Theory of Computation. Second Edition, Thomson Course Technology, Boston; 2006.
- [10] McBride B. The Resource Description Framework (RDF) and its Vocabulary Description Language RDFS, Springer, Berlin. 2004;51–65.
- [11] line G, Caroll J. W3C, Resource Description Framework (RDF): Concepts and Abstract Syntax; 2004. Available: <http://www.w3.org/TR/rdf-concepts>.
- [12] Takeuti G, Zaring W. Introduction to Axiomatic Set Theory. Springer-Verlag, Berlin; 1971. Available: <https://doi.org/10.1007/978-1-4684-9915-5>
- [13] Badder FD, Calvanese D, McGuinness D, Nardi D, Patel-Schneider P. The Description Logic Handbook. Cambridge University Press, Cambridge; 2005.
- [14] McGuinnessi D, van Harmelen F. OWL Overview, OWL Web Ontology Language Overview. W3C Proposed Recommendation 15 December 2003; 2003. Available: <http://www.w3.org/TR/2003/PR-owl-features-20031215>
- [15] Aho A. Sethi, R, Ullman, J. Compilers: Principles, Techniques, and Tools. Addison- Wesley, Boston; 1986.

- [16] Guarino N, Welty C. A Formal Ontology of Properties, Proceedings of 12-th International Conference on Knowledge Engineering and Knowledge Management in (Dieng, R. Ed.), (Berlin), Springer Verlag, Berlin. 2000; 97-112. Available: https://doi.org/10.1007/3-540-39967-4_8
- [17] Guarino N, Welty C. Evaluating Ontological Decisions with Ontoclean. Communications of the ACM. 2002;45(2):61–65. Available: https://doi.org/10.1145/503124_503150
- [18] WWelty C, Guarino N. Supporting Ontological Analysis of Taxonomic Relationship. Data & Knowledge Engineering. 2001;39:51–74. Available: [https://doi.org/10.1016/50169-023X/\(o1\)00030-1](https://doi.org/10.1016/50169-023X/(o1)00030-1)
- [19] Hurby P. Ontology-Based Domain-Driven Design. OOPSLA05 Workshop on Best Practices for Model Driven Software Development, San Diego, 2005.
- [20] Brusa G, Chiotti O, et al. A Process for Building a Domain Ontology: An Experience in Developing a Government Budgetary Ontology. Proceedings, Australian Ontology Workshop (AOW 2006). 2006;72.
- [21] Boyce S. Developing Domain Ontologies for Course Contents. International Forum of Educational Technology & Society. 2007;10:275–288.
- [22] Hernandez N. Mothe, J. Chrisment, C, Egret, D. Modeling Context Through Domain Ontology. Information Retrieval. 2007;10:143–172.
- [23] Rector A. Modularisation of Domain Ontologies Implemented in Description Logics and Related Formalism Including OWL. Proceedings of the 2nd International Conference on Knowledge Capture K-CAP-03. Sanibel Island 23-25 October, 121–128, ACM 1–5811-583-1/03/0010, October; 2003.
- [24] NCOR. National Center for Ontological Research, Domain Ontology Development Methodology; 2011.
- [25] OWL 2. Web Ontology Language Manchester Syntax; 2009. www.w3.org/TR
- [26] Horridge M. Protègè OWL Tutorial; 2011. owl.cs.manchester.ac.uk/tutorial.
- [27] The Apache Axis2/Java; 2011. Available: <http://axis.apache.org/axis2/java/core/>
- [28] The Apache CXF. An Open Source Services Framework; 2011. <http://cxf.apache.org/>
- [29] TFE Wikipedia, Enterprise Javabean; 2011. Available: http://en.wikipedia.org/wiki/Enterprise_JavaBean
- [30] Metro. Web Services for Java Platform; 2008. Available: <http://java.sun.com/webservices/reference/tutorials>
- [31] Popek G, Goldberg R. Formal Requirements for Virtualizable Third Generation Architectures. Communications of the ACM. 1974;17(7):412–421.

- [32] Rus T, Curtis D. Towards an Application Driven Software Technology. The Proceedings of the 2007 International Conference on Software Engineering Research & Practice, Las Vegas 25-28 June 2007, 282–288.
- [33] Rus T. Liberate Computer User from Programming. 12-th International Conference, AMAST 2008, Proceedings (J. Meseguer and R. G., Eds.) LNCS. 2008;5140:16—35. Springer Berlin. Available: https://doi.org/10.1007/978-3-540-79980-1_3
- [34] Monroe R. Capturing Software Architecture Design with Armani, Tech. Rep. CMU-CS-163, Carnegie Mellon University, Pittsburgh; 2001.

Biography of author(s)



Teodor Rus

Department of Computer Science, The University of Iowa, Iowa City, USA.

He was born in Romania and completed his education at "Babeş-Bolyai" University. He received research training at the Computing Institute of the Romanian Academy in Cluj, where, in 1964, he obtained the scientific title of doctor with the thesis "The Use of Tree Data Structure in Solving Non-Numeric Problems" under the guidance of Academician Tiberiu Popovici. He contributed significantly to the development of DACICC 1 and DACICC 200 computers. Starting in 1968, he served as the head of the Compiler Construction Laboratory at the Institute of Computing Technology, ITC-Cluj. In 1982, he moved to the USA and became a professor at the University of Iowa in Iowa City, where he is currently a retired Emeritus Professor. In the USA, he dedicated his entire career to Software Development, making noteworthy contributions to the use of algebraic methodology in the development of software systems. His research culminated in defining computer science as the science of using the computer to solve human problems of any nature (The computer functions as an assistant to the human mind, and its usage methodology is integrated into the user's natural language, similar to the methodology used for other utilities developed by humans to enhance their capabilities. Information Technology (IT) represents the first phase in the development of the methodology for using the computer to solve problems, where the problem domains are integrated into the computer's language. Artificial Intelligence (AI) represents a second phase in which the computer's language is integrated into the computer user's natural language. The consequence of this methodology is the simplification of computer use and the development of a computer education methodology based on the formalization of the problem domain using an ontology, in which concept terms and their meanings are accessible to both the computer and its user).

© Copyright (2024): Author(s). The licensee is the publisher (B P International).

Disclaimer:

This chapter is an extended version of the article published by the same author(s) in the following journal. Open Journal of Software Engineering and Applications, 9:52-56,2016.
Available:<https://www.scirp.org/journal/paperinformation.aspx?paperid=116394>

Peer-Review History:

This chapter was reviewed by following the Advanced Open Peer Review policy. This chapter was thoroughly checked to prevent plagiarism. As per editorial policy, a minimum of two peer-reviewers reviewed the manuscript. After review and revision of the manuscript, the Book Editor approved the manuscript for final publication. Peer review comments, comments of the editor(s), etc. are available here: <https://peerreviewarchive.com/review-history/7105E>