Computer Integration within Problem Solving Process

Teodor Rus
The University of Iowa
Department of Computer Science
Iowa City, IA 52242, USA
Email: rus@uiowa.edu

***Abstract*—In this paper we discuss a software technology that supports computer integration within the problem solving process, independent of problem domain. Due to space limitation we present here a short but comprehensive version of this research and encourage the reader to access the extended version at URL** `https://www.cs.uiowa.edu/~rus`**.**

I. 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*;
- Let the computer execute the program;
- Decode the result and extract the solution.

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 as a problem solving tool integrates the problem solving process within the computer.

Difficulties of this approach of using computers as problem solving tools result from the requirement to encode the algorithm into a program, which implies knowledge about computer architecture and functionality. Computer science experts diminish these difficulties by developing software tools which raise the machine language abstraction level towards the logical level of problem solving process. But, irrespective of their level of abstraction, these software tools represent machine computation concepts: they do not represent the concepts used by the human problem solving process. Therefore, in order to use the computer during problem solving process one needs to learn the language provided by software tools, thus increasing the level of professionalism required to computer user. However, by increasing the number and the complexity of the problem domains where computer is used as a problem solving tool, the number and the complexity of software tools required by the translation from problem domain language into the software tool language increases dramatically. Consequently, current software complexity reached a level where it threatens to kill the current computer technology itself (Horn, 2001). To tackle this situation pioneers of current software technology (Markoff, 2012) suggest "Killing the Computer to Save It".

Successes of computer use during problem solving process have evolved software tools at the level of information processing services (SaaS, 2010). Moreover, currently the networking technology allows software tools to be exchanged as standalone pieces of composeable tools called Web Services (WS). A new problem solving paradigm based on WS-s emerges, where computer based problem solving process is split between problem domain expert and computer expert according to their expertise. 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. The problems raised by the interoperability of WS-s components of SOA-s are resolved using new XML standards: Standard Object Access Protocol (SOAP), Web Service Description Language (WSDL), Universal Description, Discovery, and Integration (UDDI). 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 WS-s and SOA-s.

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 (Srinivasan and Getov, 2011) concerns mostly cloud infrastructure management, expressed in terms of Virtual Machines (VM-s) populating the cloud at a given time. But VM-s in the cloud are abstractions of computer architectures not abstractions of problem domains. Moreover, the goal of CC is stated 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 can become a problem solving tool dedicated to masses.

In this paper we show how cloud computing can be used as a mechanism that supports computer integration within the problem solving process, independent of problem domain.

The paper is organized as follows: Section II discusses the domain based problem solving process; Section III presents the domain algorithmic language, to be used by domain experts to develop domain algorithms for solving their problems; Section IV describes the process of computational emancipation of problem domains, which allows computer to be integrated within the specific problem solving process, characteristic to the domain; Section V describes the domain dedicated virtual machine, used to execute domain algorithms on the Internet, as web services; Section VI sketches the CC implementation of DAL System, the system that allows domain experts to subscribe to the cloud for computer services required by their problem domain solving algorithms.

II. 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. However, 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 $x_1,x_2 = (-b+|-\sqrt{b^2-4*a*c})/(2*a)$ 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 (Polya, 1957) 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.*

Hidden here are three things: *unknown*, *action*, *purpose*. These concepts are independent of problem domain, therefore Polya's definition is robust. Scientists solving problems manipulate the objects of their sciences whose meanings may be 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.

There are two kinds of difficulties involved in Polya problem solving methodology: difficulties that pertain to the discovery of the problem solving algorithm and difficulties that pertain to the execution of the problem solving algorithm. Algorithm discovery is characteristic to human problem solving process and due to the diversity of human range of problem domains and problems there is little mechanical help for general algorithm discovery. However, computers evolved from tools that can help performing numerical operations to tools that can perform any kind of well-defined operations. 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 (Rus and Rus, 1993):

```
beginSpec Computer
  name Hardware System is
  sort Memory, Processor, Devices, Control;
  opns receive:Device x Control->Memory;
       transmit:Memory x Control->Device;
       store:Processor x Control->Memory;
       fetch:Memory x Control->Processor;
       process:
         Memory x Processor x Control->Processor,
         Memory x Processor x Control->Memory,
         Processor x Control->Processor;
  vars PC:Control;
  axms PC.operation is receive|transmit|stores|fetch|process;
  actn PEL: while PluggedIn 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 the 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 human problem solving process, an evolving collection of programming tools have been developed as the system software. The 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 process. Current computer technology makes this task feasible by developing software tools that allow domain expert and computer expert to share the problem solving process according to their domains of expertise. Problem domain experts formulate problems and develop solution algorithms using problem domain logics. Computer experts develop software tools and provide them to computer users as WS-s. Computer network experts develop tools that allow problem solvers to ask computer networks to perform the tasks involved in their problem solving processes.

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 computer knowledge in order to consume these services.

III. Computational Language of the Problem Domain

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. Our conjecture 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 (Sipser, 2006).

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 Figure 1.

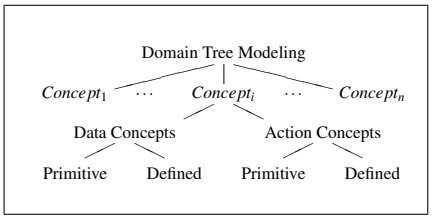

Fig. 1. Tree modeling of a problem domain

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 process and are specific to the problem solver. However, the mechanisms used to define new data and action concepts during problem solving process 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 and 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. These are valid expressions in the natural language of the problem solvers, which are understood by all domain experts because they use only concepts familiar to the domain experts.

This domain modeling implies that the solution (algorithm) of any domain problem defines a new characteristic concept of the 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.

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.

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. Grammatically, the initial terms of the DAL vocabulary would be categorized as nouns, verbs, adjectives, and adverbs. The statement formation rules are chosen to fit the Resource Description Framework (RDF) used by the Semantic Web (Kline and Caroll, 2004), $Subject \xrightarrow{Action} Object$, $Subject \xrightarrow{Property} Object$, where $Subject$, $Object$, $Action$, $Property$ are elements of the DAL vocabulary. 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 (Takeuti and Zaring, 1971).

Computational nature of DAL is obtained by DAL's se-

mantics specification using a description logic (Badder et al., 2005) whose model is defined as follows:

- Implement every concept C of the DAL terminology as a $WS(C)$. Let URI(C) be the URL of the WS(C).
- Implement formation rules $Subject \stackrel{Action}{\longrightarrow} Object$ by WS(Action) whose input and output are elements of $Subject \times Object$.
- Implement formation rules $Subject \stackrel{Property}{\longrightarrow} Object$ by WS(Property) that input tuples of $Subject \times Object$ and return true or false.
- Implement every solution algorithm by a WS obtained by the composition of the web services employed in the algorithm.

Further, structure DAL and its model using a domain ontology represented by a file in the Web Ontology Language, (OWL) (McGuinness and van Harmelen, 2003). 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).

IV. Computational Emancipation of a Problem Domain

The DAL algorithm execution discussed in Section III 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 process 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 during problem solving process 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 III and can be achieved by:

1) Software tools to automate the process of domain ontology creation and implementations using the OWL(D);
2) Software tools that automate WS generation and optimize the search for the concept implementation in OWL(D) 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). In the extended version of this paper (Rus, 2012) we show how few of such tools (Axis, Metro, Protégé, RDF, OWL, etc.) are used.

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 (Guarino and Welty, 2002; Welty and Guarino, 2001) 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 (Rector, 2003). Domain ontology construction is a two-step process performed by a collaboration between domain expert and computer expert. First, a Domain Expert Ontology (DEO) is built (in the cloud) which represents the primitive concepts of the domain available to all domain experts. Second, each problem solver inherits DEO and extends it automatically with the domain concepts she learns during her own problem solving process, thus developing a User Own Ontology (UOO). The process is performed by appropriate tools, as explained in (Rus, 2012).

V. 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 DAL algorithm 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 DAL algorithm and the domain ontology. The result can be seen as a "program" in the language of the brain assistant used by problem solver to execute the DAL algorithm. Since the operations performed by this automaton 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.
- 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 II) 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 algorithn)
Extract result + dysplay it to the user
```

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 a DAL algorithm $\mathcal{A}$ into an expression tree $ET(\mathcal{A})$ 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 interpretor is implemented by a stack machine similar to Java Virtual Machine (JVM).

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) (Rus and Curtis, 2007; Rus, 2008).

SADL has been conceived as a language suitable to describe functional behavior of component-based software architectures, where components are standalone and composeable pieces of software.

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.

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 nulary 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 namespace implementing the ontology of an arithmetic domain.

The composed constructs of the SADL language are XML constructs composed with the terms: `foreach`, `if`, `ifthen`, `next`, `etc`. Example, the SDAL 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" />;
```

SADL interpreter inputs a SADL expression and interpret 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 represents a process executed on Internet, the flow of control during a SADL expression evaluation requires the synchronization of these processes. Thus, the SADL interpreter performs a distributed implementation of the the DAL algorithm. The simplest synchronization mechanism used to control the flow of processes performing a DAL algorithm is provided by a `(wait, signal)` 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 executing a DAL algorithm to perform in parallel.

VI. DAL System

Cloud-implementation of the DAL System is described in Figure 2. 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 maintains a data base where all DEO-s of the CEADed domains 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 Figure 2, 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 my

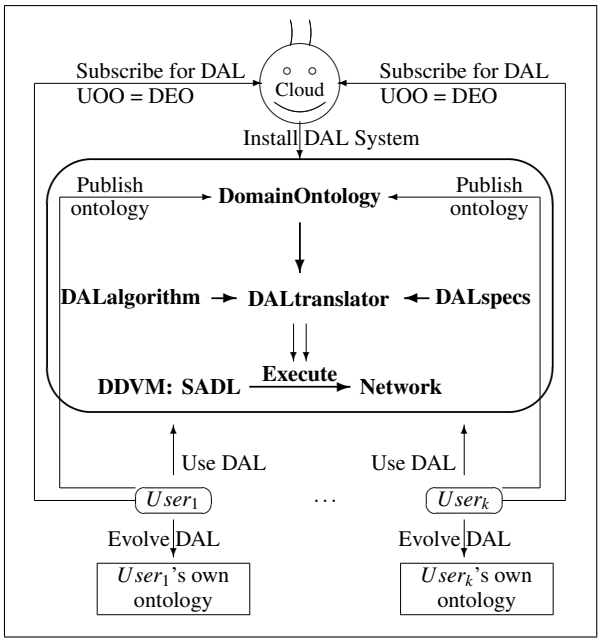

Fig. 2. Architecture of DAL System

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.

A user doesn't need a computer to interact with the DAL System. An iPad which provides a two-way communication using a command language can be used in this purpose.

VII. Conclusion

The research reported in this paper shows that software development for non-expert computer user open an unlimited area for computer technology development. This has the potential to empower human being with the computer as a brain tool (oracle). To achieve this potential computer use needs to be freed from its `one-size-fits-all` pattern, and let it be, as it has proven to be, a problem solving tool that may act in any problem domain as a `domain oracle`.

Acknowledgment

The author thanks here generations of graduate students at the University of Iowa, Department of Computer Science, who developed the TICS software tools used in the design and implementation of DAL System. Special thanks are due to Cuong Bui who implemented the proof of concept for arithmetic domain, that is available (upon request) at URL `bula1.cs.uiowa.edu`.

References

Aho, A., Sethi, R., and Ullman, J. (January 1, 1986). *Compilers: Principles, Techniques, and Tools*. Addison Wesley.

ApacheCXF (2011). `http://cxf.apache.org/`.

Axis/Java (2011). `http://axis.apache.org/axis2/java/core/`

Badder, F., Calvanese, D., McGuinnes, D., Nardi, D., and Patel-Schneider, P., editors (2005). *The Description Logic Handbook*. Cambridge University Press.

Guarino, N. and Welty, C. (2000). A formal ontology of properties. In Dieng, R., editor, *Proceedings of 12-th International Conference on Knowledge Engineering and Knowledge Management*, Berlin. Springer Verlag.

Guarino, N. and Welty, C. (2002). Evaluating ontological decisions with ontoclean. *CACM*, 45(2):61–65.

Hernandez, N., Mothe, J., Chrisment, C., and Egret, D. (2007). Modeling context through domain ontology. *Inf Retrieval*, 10:143–172.

Horn, P. (2001). Autonomic computing: IBM's perspective on the state of the information technology. http://www.research.ibm.com/autonomic/manifesto.

Horridge, M. (2011). Protègè-owl tutorial. http://owl.cs.manchester.ac.uk/tutorials/protegeowltutorial/.

Hruby, P. (2005). Ontology-based domain-driven design. In *OOPSLA05 Workshop on Best Practices for Model Driven Software Development*, San Diego, CA.

Kline, G. and Caroll, J. (2004). W3C, Resource Description Framework (RDF): Concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/.

Markoff, J. (2012). Killing the computer to save it. *ACM TechNews*, October(31).

McBride, B. (2004). *The Resource Description Framework (RDF) and its Vocabulary Description Language RDFS*, pages 51–65. Springer.

McGuinness, D. and van Harmelen, F. (2003). *OWL Overview, OWL Web Ontology Language Overview. W3C Proposed Recommendation 15 December 2003*. http://www.w3.org/TR/2003/PR-owl-features-20031215/.

Metro (2008). Web services for java platform. `http://java.sun.com/webservices/reference/-/tutorials/index.jsp`.

Monroe, R. (2001). Capturing software architecture design with armani. Technical Report CMU-CS-163, Carnegie Mellon University.

OpenStructs, TechWiki (2011). Lightweight, domain ontology development methodology. `http://techwiki.openstructs.org/index.php/`

OWL2 (2009). OWL2 Web Ontology Language Manchester Syntax. `http://www.w3.org/TR/owl2-manchester-syntax/` .

OWL2 Primer (2009). OWL2 Web Ontology Language Primer. `http://www.w3.org/TR/owl2-primer/` .

Polya, G. (1957). *How To Solve It*. Princeton University Press, second edition.

Popek, G. and Goldberg, R. (1974). Formal requirements for virtualizable third generation architectures. *CACM*, 17(7):412–421.

Rector, A. (2003). Modularization of domain ontologies implemented in description logics and related formalism including owl. In *Proceedings, K-CAP-03*, pages 121–128. ACM 1–5811-583–1/03/0010.

Rus, T. (2008). Liberate computer user from programming. In Meseguer, J. and G., R., editors, *12-th International Conference, AMAST 2008, Proceedings*, volume LNCS 5140, pages 16–35. Springer.

Rus, T. (2012). Computer Intergration within Problem Solving Process. `https://www.cs.uiowa.edu/~rus`.

Rus, T. and Curtis, D. (2007). Towards an application driven software technology. In *The proceedings of the 2007 International Conference on Software Engineering Research & Practice*, page 282288, Las Vegas, NV, USA.

Rus, T. and Rus, D. (1993). *System Software and Software Systems: Concepts and Methodology*. World Scientific.

SaaS (2010). Software as a Service (SaaS). `http://en.wikipedia.org/wiki/Software_as_a_service`.

Sipser, M. (2006). *Introduction to the Theory of Computation*. Thomson Course Technology, second edition.

Srinivasan, N. and Getov, V. (2011). Navigating the cloud computing landscape – technologies, services, and adopters. *Computer*, 44(3). IBM and University of Westminster: Cloud Computing: Infrastructure-As-A-Service, Platform-As-A-Service, Software-As-A-Service.

Takeuti, G. and Zaring, W. (1971). *Introduction to Axiomatic Set Theory*. Springer-Verlag.

Welty, C. and Guarino, N. (2001). Supporting ontological analysis of taxonomic relationship. *Data & Knowledge Engineering*, 39:51–74.

Wikipedia, T. F. E. (2011). Enterprise javabean. `http://en.wikipedia.org/wiki/Enterprise_JavaBean`.