Toward Application Driven Software Technology

Teodor Rus
rus@cs.uiowa.edu
University of Iowa
Iowa City, IA 52242

Donald Ephraim Curtis
donald-curtis@uiowa.edu
University of Iowa
Iowa City, IA 52242

Keywords: ontology, computational emancipation, process semantics

Abstract

Current problem solving methodology by computers requires application domain expert to develop programs in a programming language. Software tools designed to ease this task and the expansion of computer use in practically all aspects of human life lead to an increasing software complexity. In this paper we present a solution to the problems resulting from increasing software complexity by developing a methodology for problem solving with the computer where problem solving process is split between application domain expert and software expert as follows: (1) AD expert develops solution algorithms using the natural language of the application domain; (2) IT expert implements universal algorithms that characterize the application domain; (3) the AD is provided with an interpreter that implements AD algorithms in terms of the processes implementing the components of the AD algorithms. This is realized by Computational Emancipation of the Application Domain*, (CEAD). Here we illustrate these ideas using linear algebra as application domain.*

1. Application-driven problem solving methodology

Throughout the problem solving process, problem solvers manipulate concepts from their domain of interest (problem domain) to form problem solutions. The invention of computers allowed the problem solving process to evolve so that computers could be used as problem solving tools [Scr97], [LM97]. We now see problem solutions as complex processes which can be carried out by the digital computer. But current methodology of computer use for problem solving has also created a gap within the problem solving process between problem domain experts (who formulate problems and their solutions) and information technology (IT) domain experts (who implement solutions).

Domain experts use languages specific to their application domains (AD) to represent problems and their solutions [Jon00]. These AD languages are very different from the languages used to represent machine computations (programming languages). The vocabulary of an AD language is built on conceptual abstractions that characterize the domain and the AD language is used in the cognition process. That is, AD language expressions represent AD knowledge. For any domain each term in its AD language has a well-defined meaning for all domain experts within that domain. The vocabulary of a programing language is built on concepts representing computer resources (memory, processor, disk, etc.) and their expressions represent computations to be carried out by the machine. That is, programming language expressions (i.e., programs) represent machine-encoded computations so that they can be performed by the machine on which that programming language is implemented. The differences among the languages of an AD and the IT are related to the computational differences between the human brain and digital computers. Computers perform operations based on syntax, differing from the human brain which performs operations cognitively, based on semantic relations between terms used [Sea90]. While a computer can perform complex computation much faster than the brain, the language that problem solvers use to express their solutions is *not necessarily* the language that is used to implement these solutions in the IT domain. This gives the problem solver two options:

1) Gain enough knowledge about the IT domain to implement their problem solutions. The

concern here is that the domain expert must become an expert in the IT domain or risk poor implementations.

2) Employ the help of an IT expert and attempt to express the problem solution in a common solution representation understood by both the domain expert and the IT expert. With this option the problem solver views the implementation as a black box that delivers the problem solution.

Computer-based problem solving methodology has evolved thus far following mostly the first option mentioned above. Problem solving process is helped by appropriate software (high-level programming languages and tools that map these languages into machine computations, such as compilers and interpreters) developed by IT experts. But the expansion of computer use in practically all aspects of human life led to the development of complex software, whose complexity grows with the application domain, that can hardly be manipulated by application domain experts. Some IT experts believe that ubiquitous computing [LY02] implies the extension of computational thinking [Win06] to all AD characterizing human activity. Since one cannot expect that all humans become IT experts, the IT domain is stressed by the requirement to produce appropriate problem solving tools for all aspects of human life.

A viable solution to the problem created by growing software complexity with problem domain expansion has been sought from the very infancy of software development. The most notable such solutions have been developed by using domain-specific programming languages (DSL) which tailor programming language to the application domain [MJS05]. DSL offers gains in expressiveness and ease of use compared with general-purpose programming languages, but DSL are still programming languages that require their users to have familiarity with computational thinking and computer platforms implementing them. In other words DSL (as generals-purpose programming languages) are problem-solving tools that handle computer abstractions, not AD abstractions, and thus are natural for IT experts, not for AD experts. On the other hand looking at other technologies put in the human service one may observe that their evolution does not require their user to become their expert. For example, a car driver is not required to become a mechanical engineer. This means that for further developments in computer technology we need to approach the second option for computer use to solve human problems and to develop an alternative computer-based problem solving methodology which does not require the computer user to become an IT expert.

The premise of such a new computer-based problem solving methodology is the bridging of the semantic gap between AD experts and IT experts created by the current methodology. To understand why this gap cannot be bridged by the programming language semantics used so far, we sketch here the three main mechanism used to assign meaning to programming language constructs: operational semantics, denotational semantics, and axiomatic semantics.

With operational semantics the meaning of a language construct is described in terms of an abstract machine which has a set of states and primitive operations. One of the most notable such abstract machines is a "Turing Machine". The benefit of this approach is that computational contents of a language construct can be represented in such a simple manner that the meaning of such a construct cannot be misunderstood. However, this method does not bridge the gap between the AD and IT domains. Rather, it adds a new level of abstraction in the communication which requires both domain experts to be familiar with. AD expert needs to handle both programming language and abstract machine language, IT expert is required to translate the programming language and the abstract machine language into the language of the machine performing computations. Denotational semantics provides a similar approach where language meaning is obtained by associating language constructs with well defined mathematical objects called *denotations*. So, rather than describe the computational contents of language constructs in terms of machine operations performing them, denotational semantics associates language constructs with functions which convert syntactical constructs to the abstract values they represent [Sto77]. Again, though this approach provides an accurate way to represent meaning, it now requires both AD and IT domain experts to become experts of the mathematical domain used as denotations. Axiomatic semantics associates axioms with language constructs where axioms are assertions about the state before and after the computation represented by the construct is performed [Hoa69]. However, to handle axiomatic semantics AD and IT experts are required to handle mathematical logic. The use of axiomatic semantics suffers from the same symptoms as operational and denotational semantics, requiring the user to learn the logical language used to specify these axioms. Hence, in all cases programming language semantics augment expert communication with a new layer of abstraction which can only deepen the

semantic gap.

With the new problem-solving methodology we envision [RC06] the AD expert and IT expert collaborate to solve problems with a computer by a new protocol:

1) AD experts formulate problems and develop solution algorithms using the natural language of their application domain;
2) IT experts implement computation processes represented by universal stand-alone algorithms that characterize application domains;
3) AD concepts are associated with universal stand-alone algorithms characteristic to the application domain, called components.

Note: because concepts used in AD algorithms are associated with universal algorithms they can be implemented by IT experts independent of their usage in solution algorithms. Moreover, because these algorithms are stand-alone components characteristic to the application domain, AD algorithms using them can be executed by interpretation rather than by translation thus avoiding the communication gap between AD and IT experts.

The implementation of this protocol of communication allows AD and IT to evolve independently and in parallel. The association of AD concepts with computer artifacts that complies with the collaboration protocol sketched above and facilitates software development that supports algorithm implementations by interpretation requires that both AD and IT be appropriately organized. With the application driven software that demonstrates this idea [RC06] we used the domain ontology to structure AD and Semantic Web Languages to develop the software that implements AD algorithms. Consequently we introduce the term *Computational Emancipation of the Application Domain* (CEAD) where each term representing a domain abstraction is associated with the computer artifact implementing it in the IT domain providing *process semantics*. CEAD is not a programming language and consequently is not a DSL. CEAD implies the following tasks:

(i) Structure the application domain using an appropriate ontology.
(ii) Associate each concept in the ontology obtained at (i) with appropriate computer artifact implementing it on a computer platform.
(iii) Interpret the AD algorithms expressed in the natural language of the domain by composing the processes associated with the components of the algorithms.

The evolutionary nature of the domain ontology and the potential distribution of the computer resources used to implement the components of AD algorithms on computer network support the implementation of AD algorithms by mapping them first into an appropriate interpretable language. The language SADL [RC06] has been designed with this goal in mind. The unit of computation in SADL is the process associated with the ontology concepts used in the algorithm; the structure of the AD algorithm in SADL is a composed process whose components are either processes associated with algorithm components or are processes used by SADL interpreter to compose processes. Since computer artifacts used in CEAD as semantics are resources identifiable by URI-s [TFIM98] we express processes in SADL using XML elements whose tags are process components and whose attributes specify completely these processes in terms of their input/output behavior. Thus SADL can be seen as a DSL where the domain is similar to RDF [McB04]. In Section 2 we illustrate CEAD with the linear algebra and SADL with the SADL expression of Gaussian elimination algorithm for solving systems of linear equations. To increase the readability of a SADL-expression we assume that software artifacts used to emancipate a domain ontology are collected into an URI-table and are referenced in the SADL expressions of the domain expert algorithms using the notation `URI(ontology-concept)`. For example, if A is a matrix then `URI(A)` is the URI of a file that implement the matrix A.

CEAD opens the door for domain experts to work in their own language using domain terminology while developing solutions that represent computational process which can thus be performed by using the computer as a tool. Using the concept of computational emancipation we revisit the problem solving process and consider a new methodology:

1) Domain experts develop problem solutions using domain terms which can be interpreted directly from their domain expressions.
2) IT experts develop tools which map domain expressions to IT implementations using the computationally emancipated domains.
3) There is a beneficial feedback between AD and IT where AD benefits from IT advances in persistent information structuring and IT benefits from AD enriching its problem solving environments with universal algorithms that characterize the AD.

2. Computational emancipation of AD

Ontology is the philosophical study of what *is* [Qui69], concerned with identifying and categorizing

those things which exist and how those things relate to each other. Ontology found its way into computer science as the "specification of a conceptualization" through the work of Gruber [Gru93].

Description logics (DL) is a formalism that evolved from knowledge representations used in Artificial Intelligence research. More recently this formalism is used for formal specification of ontologies used with the semantic web [MvH]. That is, DL provides the mathematical machinery to formally represent ontologies in such a way that they can be computationally reasoned about. Using a DL language we can define formally the ontology of an AD as follows. For a given domain we have a collection of terms $D = \{t_1, t_2, \dots\}$ representing basic (or primitive) concepts of the domain, a set of relations $R = \{R_1, R_2, \dots\}$ (called roles) representing fundamental properties of the domain concepts, and a set of constructors that can be used to define terms representing new concepts of the domain, $C = \{C_1, C_2, \dots\}$. Constructors allow domain concepts and roles to be layered on a *sub-class/super-class* hierarchy, thus providing the framework for inductive and compositional specification of the domain ontology. For a set-theoretic model of the DL, terms are associated with unary predicates satisfied by the sets of objects in the universe of discourse (UoD), roles are associated with binary relations representing properties of the (UoD), and constructors are associated with logic operators on the UoD. In addition, by computational emancipation of the domain, terms are associated with computer artifacts (representing computational processes) implementing them, roles are associated with properties of such process, and constructors are associated with process composition operators. Thus, the set-theoretic model of the DL allows us to reason about domain objects while the process-model of the DL allows us to implement computation algorithms representing problem domain solutions by process composition rather than language translation. Figures 2,3,4 illustrate this idea using linear algebra as the AD and tree representation of the ontology.

Notations: ∇ denotes the operator that maps a matrix into its diagonal form, Δ denotes the operator that computes the determinant of a square non-singular matrix, and $\circ$ denotes matrix concatenation with a vector.

In real applications, the computational emancipation of an application domain is an operation performed on knowledge represented using description logics where we augment each term $t \in D$ with a uniform resource identifier (URI) that points to the computer artifact which implements the concept represented by that term. We denote an emancipation of a domain D as E_D where if $(t, u) \in E_D$ then $t \in D$ and u is the URI of the computer artifact implementing the concept t. We say that a domain is *fully emancipated* iff for all $t \in D$, there is a computer artifact a such that $(t, a) \in E_d$. Each computer artifact is one of the following;

1) a persistent data representation, such as a files
2) a transient process that implements a domain concept by an expert solution thus solving a problem;
3) the value produced by a process that implement a domain concept;
4) the expression of a repeatable process that represents the implementation of a new concept thus evolving the domain ontology.

Computational emancipation allows the domain expert to perform problem solutions by process composition without requiring translation into computer implementations. That is, computational emancipation allows us to map domain expressions into processes that execute their IT implementations [RC06]. Consider a sequential solution S in the fully emancipated domain D, $S = P(t_1, \dots, t_n)$, where P is a DL expression and $t_i \in D$ for $1 \leq i \leq n$. In other words, the solution S is a process resulting from the composition of processes associated with t_i, $1 \leq i \leq n$, according to the composition of the operators specified by P. Because D is fully emancipated, for each t_i there exists u_i such that $(t_i, u_i) \in E_D$, for $1 \leq i \leq n$, which can perform the process represented by t_i and thus we can execute the solution represented by S. For example, using the ontology in Figure 2, a mathematician expresses the algorithm for solving linear equation systems by Gaussian elimination as follows:

```
Gaussian Elimination:
  Input: {Matrix A, Vector B}
  Output: Vector X
    M := Concatenate (A, B)
    F := FowardElimination (M)
    X := BackwardElimination (F)
```

Further, the AD (linear algebra in our example) is provided with a translator that uses AD ontology to map AD algorithms into SADL expressions thus facilitating their execution by interpretation. This is illustrated in Figure 1 by the SADL expression of the Gaussian elimination algorithm. The SADL expression thus obtained is further mapped by SADL interpretor into the process that performs the AD algorithm by interpretation.

```
<?xml version="1.0" ?>
<sadl>
  <system name="URI(GaussianElimination)" input="URI(A) URI(B)" output="URI(X)">
    <component name="URI(Concatenate)" input="URI(A) URI(B)" output="URI(M)" />
    <component name="URI(ForwardElimination)" input="URI(M)" output="URI(F)" />
    <component name="URI(BackwardElimination)" input="URI(F)" output="URI(X)" />
  </system>
</sadl>
```

Figure 1. SADL expression of Gaussian Elimination

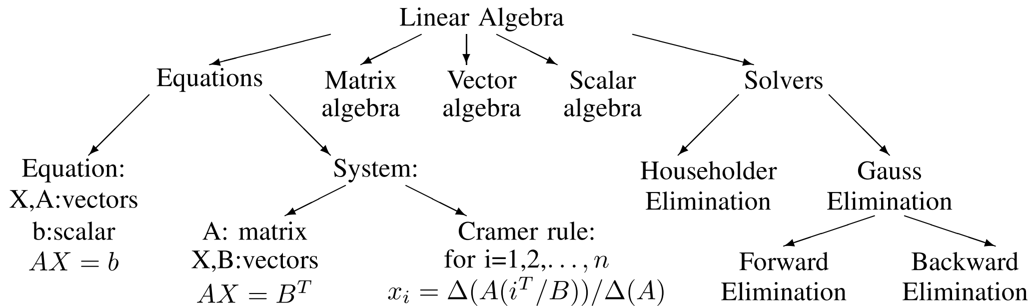

Figure 2. A fragment of linear algebra ontology

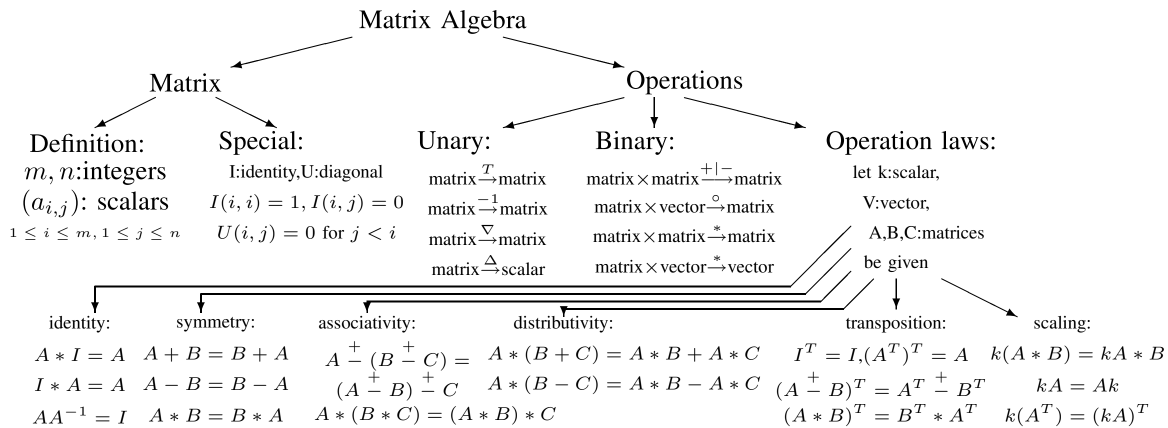

Figure 3. A fragment of matrix algebra ontology

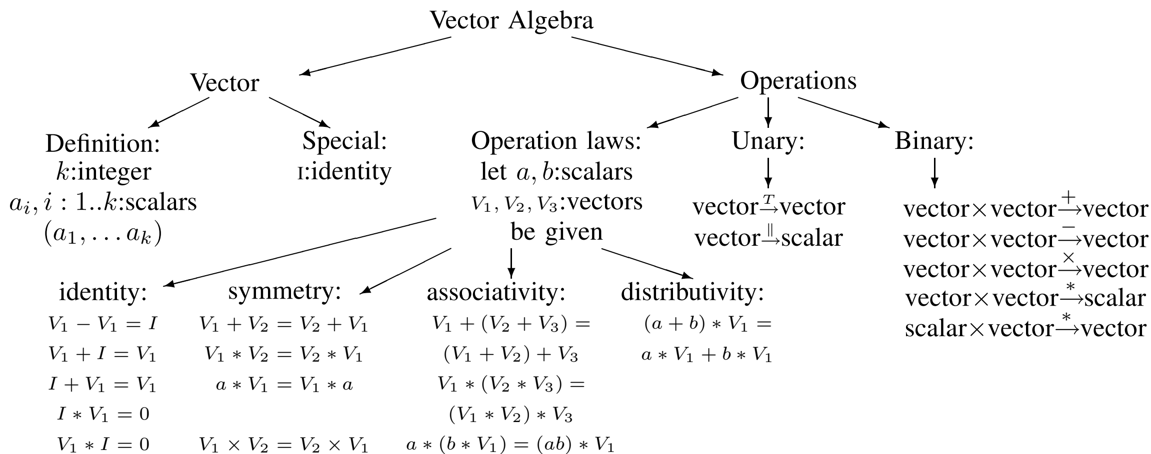

Figure 4. A fragment of a vector algebra ontology

3. Impact of CEAD

Allowing domain experts to formulate and solve problems within their domain of discourse impacts the problem solving process both inside and outside the computer science discipline. We look here at the impact of this problem solving methodology on the areas of software engineering, global collaboration on problem solving projects, and teaching and learning computer technology. To illustrate this impact we first examine the problem solving concerns in each area and then show how computational emancipation of the problem domain relaxes these concerns.

Software engineering requires the customer, problem solver, to explain their solutions to the software engineer, IT expert. While the Internet has allowed both parties to be separated geographically, the semantic gap between the problem domain and the IT domain remains the same. The fundamental difficulties in their collaboration are still determined by the communication [Par06]. The software engineer, concerned only with the resulting implementation, views the software engineering process independent of the task performed by the tool thus developed in the universe of discourse. On the other hand the client expects that the tool thus developed fits within her universe of discourse. This seldom happens. Software engineering research is dominated by approaches aimed to bridge this gap. Two such approaches are model driven architectures [SK97], [GBI+04], [Coo04], [BGK+06] and domain specific languages [GM03], [MJS05]. Model driven architectures use computerized representations [Coo04] of elements which correspond to concepts in the problem domain. Consequently, model driven architectures structure their implementations using these IT representations and not the domain concepts. Domain specific languages are designed so that the programming language more accurately represents the problem domain. While this makes it easier for software engineer to handle domain concepts, it does not directly benefit the domain expert trying to implement their solution. Computational emancipation of the problem domain addresses this problem by relieving the dependency of the software engineer on the problem domain and of the problem solver on computerized representation. The software engineer can focus her efforts on computational emancipation of the IT domain, developing tools to help the interpretation of AD algorithms. The problem solver can focus her efforts on computational emancipation of application domain, supplying the software engineering with new universal algorithms used as components in her solutions.

Collaboration among people solving problems is a major enterprise in modern society [CS99]. While collaborators share domain concepts and solutions their computer implementations often differ, leading to a continuous process of "reinventing the wheel". In other words, one domain solution may get reimplemented numerous times by different researchers requiring that same computer implementation. We see this in computational chemistry where there exist numerous quantum chemistry packages, DALTON [DLT] and GAMESS [GMS] being two of them, which contain completely different implementations of the same domain solution. This means that application directed collaboration breaks down at the IT level and slows research progress. By computational emancipation of the problem domain involved in collaboration we connect domain concepts directly to the computer artifacts which implement them. This allows the collaboration within domains to continue independent of actual implementations. That is, with computational emancipation of the problem domain collaboration extends to process, not just the language.

The problem of teaching computer technology is directly related to the obstacle of learning. Historically we inherited text-book based teaching where the student and teacher interact through a given text, the textbook, that describes the domain of discourse. But often computer gadgets are developed faster than the textbook describing them can be available. Thus the textbook becomes obsolete at its arrival. With computational emancipation of the application domain both teacher and student can share knowledge by hands-on the concepts, and their computer implementations, at the application domain level, independent of textbook describing their implementations. Thus, learning within a domain of interest can be concerned strictly with understanding the concepts within that domain. With current teaching methodology, as pupils learn a subject within an application domain (say linear algebra) they are simultaneously required to become familiarized with domain concepts (such as system of equations) and computer concepts (data structures and functions) implementing them. As a result, students can relate domain concepts only by expressing them in terms of IT concepts. This requires one domain to be prerequisite of another and thus in addition to the difficulties created by indirect learning, this creates delays in the process. Rather than blending AD and IT domains together, computational emancipation aims to separate these domains and bridge the semantic gap between the subject and its implemen-

tation. Additionally, CEAD directs the design and implementation of mindtools [JC00] toward learning domain structuring which improves both efficiency and effectiveness of computers as cognitive tools in the classroom.

References

[BGK+06] K. Balasubramanian, A. Gokhale, G. Karsai, J. Stipanovits, and Sandeep Neema. Developing applications using model-driven design environments. *Computer*, February:33–41, 2006.

[Coo04] S Cook. Domain-specific modeling and model driven architecture. *MDA Journal*, January 2004. Available at: http://www.bptrends.com/search.cfm?keyword =MDA+Journal&gogo=1,2004.

[CS99] I. Chompalov and W. Shrum. Institutional collaboration in science: A typology of technological practice. *Science, Technology, & Human Values*, 24(3):338–372, 1999.

[DLT] Dalton quantum chemistry program. http://www.kjemi.uio.no/software/dalton/ dalton.html

[FMO+96] M.P. Fiore, E. Moggi, P. O'Hearn, J. Riecke, Rosolini G., and I. Stark. Domains and denotational semantics: History, accomplishments and open problems. *Bulletin of EATCS*, 59:227–256, 1996.

[GBI+04] B. Grady, A. Brown, S. Iyengar, J. Rambaugh, and Selic. B. An MDA manifesto. *MDA Journal*, May 2004. Available at: http://www.bptrends.com/search.cfm?keyword =MDA+Journal&gogo=1,2004.

[GM03] M. Grüninger and C. Menzel. The process specification language (psl) – theory and applications. *AI Magazine*, 24(3):63–74, 2003.

[GMS] The general atomic and molecular electronic structure system (gamess). http://www.msg.ameslab.gov/GAMESS/

[Gru93] T.R. Gruber. A translation approach to portable ontology specification. *Knowledge Acquisition*, 5(2):199-220, 1993.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[JC00] D.H. Jonassen and C.S Carr. *Mindtools: Affording Multiple Knowledge Representations for Learning*, pages 165–196. Lawrence Erlbaum Associates, Publishers, 2000.

[Jon00] D.H. Jonassen. Toward a design theory of problem solving. *Educational Technology Research and Development*, 48(4):63–85, August 2000.

[LM97] R.H. Landau and M.J.P Mejia. *Computational Physics: Problem Solving with Computers*. Wiley-Interscience, 1997.

[LY02] K. Lyytine and Y. Yoo. Issues and chellenges in ubiquitous computing. *Communications of the ACM*, 45(12), 2002.

[McB04] B. McBride. *The Resource Description Framework (RDF) and its Vocabulary Description Language RDFS*, pages 51–65. Springer, 2004.

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

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

[Par06] D. Parnas. Agile methods and gsd: The wrong solution to an old but real problem. *Communications of the ACM*, 49(10):29–30, 2006.

[Qui69] W.V. Quine. *Ontological relativity and other essays*. Columbia University Press, 1969.

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

[Scr97] G.W. Scragg. *Problem Solving with Computers*. Jones and Bartlett, 1997.

[Sea90] J.R. Searle. Is the brain a digital computer? *Proceedings and Addresses of the American Philosophical Association*, 64(3):21–37, November 1990.

[SK97] J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE Computer*, pages 110–112, April 1997.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[TFIM98] Berners-Lee T., R. Fielding, U.C. Irvine, and L. Masinter. Rfc 2396: Uniform resource identifiers (uri): Generic syntax, 1998.

[Tur50] A. Turing. Computing machinery and intelligence. *Mind*, 59:433–460, 1950.

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