Integrating Computing Education with Ontology Engineering

Teodor Rus[1], PI, Department of Computer Science
Marc Armstrong[1], CoPI, Department of Geography
George Constantinescu[1], CoPI, Department of Civil and Environmental Engineering
Michael Denny[2], CoPI, Principal Ontologist

June 6, 2005

[1] The University of Iowa, Iowa City, IA 52242
[2] Concurrent Technologies Corporation (CTC), Johnstown, PA 15904

Contents

Project Summary

Software systems so far have evolved as problem-solving tools independently of their problem solving methodology. With these tools, problem solving is specified through program development using a conventional programming language. The consequences are the absence of domain-oriented abstraction in the problem solving process and an exponential increase in software complexity. Our proposal seeks research and education funds to support the development of a domain-oriented ontology-based approach to problem solving that integrates the process of computer education with domain-ontology engineering. The objectives of the project are:

- Develop the methodology for application-domain structuring using ontology engineering tools and design software architecture description languages (ADL-s) that can capture problem-solving process as ADL expressions using domain ontology;
- Implement ADL interpreters that integrate computing abstractions (used in ADL expressions) and generate computing processes (performing the algorithms expressed by ADL expressions);
- Use this methodology to conduct problem solving experiments in such diverse problem domains as Internet Agents, Geography, and Hydrology; integrate this problem solving methodology within the computing education by developing and teaching hands-on-experience courses.

These objectives are accomplished by a *process-based application development system (PADS)* where computer users develop architectures of their problem solving systems while the PADS maps these architectures into processes that perform solution algorithms. No programming as usual is involved. With this methodology, domain professionals develop domain-ontologies and demonstrate the use of the PADS by teaching courses using hands-on high performance computing. System software professionals develop (and use existent) tools that support domain-ontology integrations and reasoning about software systems thus developed. Computer users manipulate domain oriented ontological terms representing *specification mechanisms, tools*, and *components*. We will illustrate this within an Interdisciplinary Computing Laboratory created as a test-bed for this methodology.

Intellectual merit: The fundamental idea of this proposal is the development of a computerized education based on a hands-on problem solving teaching methodology using problem-domain structuring by ontology engineering tools. This leads to a true separation of software system architecture from software system functionality and to the development of component-based software systems by *composing processes performed by the component codes rather than composing codes*. The use of the computation process as a unit of composition during software system development opens the way for using natural language as a logical tool in the process of problem solving with a computer. The consequence is the domain-expertise reusability along with the code reusability.

Broader impacts: Our research aims at using domain ontology as semantic characterization of the software architecture description languages. The teaching approach we pursue is a component of the problem solving methodology developed in our previous research. This provides a framework to create domain-oriented problem solving environments (PSE) that integrate research and education while promoting teaching and learning. PSEs created by students during their teaching/learning period become the generators of enhanced PSEs in the domain of activity in which they are later employed. The machine independent paradigm for software development based on expertise reusability and the embedding of domain-oriented PSEs into computing network guarantees further impact of the results we will produce.

1 Project description

Computer usage in science, technology, and everyday life, has exploded during the last two decades. One side-effect of this explosion is the exponential growth of software system complexity. Consequently, the cost of developing, maintaining, and using software systems for problem solving has become exorbitant. Researchers from IBM [Hor] believe that in order to sustain the current approach of problem solving with a computer, given the growth rate of computer applications, fifteen years from now the entire US population would need to work on software installation and maintenance. Among the major causes of this situation seem to be the conventional approach of computer usage for problem solving. Current methodology for computer usage requires that the human logic of problem solving be encoded into a program in computer memory whose execution is described by the loop:

$$\texttt{while((PC).Opcode} \neq \texttt{Halt)\{Execute(PC); PC := Next(PC)\}}$$

Here `PC` is the program counter register holding the memory address of the current instruction of the program, `Execute(PC)` performs the operation encoded in the current instruction, and `Next(PC)` determines the address of the next instruction of the program. Computer systems set this loop as the foundation for their usage in problem solving.

1.1 Problem formulation

To handle software system complexity research funding agencies have promoted the creation of the *science of design* [Fre04]; the computer industry, [IBM], suggests the development of autonomous computers, whose behavior mimics the behavior of the human body; academic research, [GHR94, HRGB00, PMJ+05], suggests the development of domain-oriented Problem Solving Environments, PSEs; the software industry, [BSR03, Bax, Big98a, Nei80, SK97, Cor] suggests the automation of program development and implementation. We believe that in order to subdue the complexity of current software, computing education needs to join funding agencies, computer industry, academic research, and software industry for the development of a *computer supported problem solving approach based on the human logic of the problem domain and on the appropriate domain-oriented problem solving methodology*. For this to happens, one must first observe that problem domain and computer programming are two different aspects of the computerized problem solving methodology: problem-domain offers the logic for problem specification and the abstractions required for the development of solution algorithms; computer programming offers the tools to implement solution algorithms. Software systems should provide mechanisms capable to map domain-oriented solution algorithms into computer processes performing these algorithm, thus short-circuiting the phase of program development. While computers should continue to perform programs, problem solvers should not be required to develop such programs. Rather, problem solvers should be required to develop solution-algorithms by manipulating abstractions characterizing their problem-domains. This is already well illustrated by problem solving within system-software domain. Programmers do not write compilers to map their programs in machine-language thus making them executable; programmers use existing compilers to perform this task. Computer operators do not manipulate hardware resources while executing programs; they express program execution using control-languages and use operating systems to manipulate hardware and software resources involved in these programs. System software architects creating software architecture description languages (such as Acme[GMW02]) do not manipulate actual software components while creating

software architectures; they manipulate a domain-ontology[1] whose vocabulary contain such terms as *component, connector, port, role, representation, systems, style,* using properties and the first order logic, FOL, to express their software architectures and reason about their properties [Mon01]. Why mathematicians (or, as a matter of fact any other computer user) involved in solving, say linear-equation or partial differential equations, or any kind of optimization, cannot use a similar methodology? They should be able to describe the activity involved in their problem solving using an appropriate domain-oriented language and let the tools provided in the system software map these descriptions into computer processes solving their problems. Computer users (as any other kind of technology user) should not be required to master computer science domain of expertise. Car-drivers, for example, are not required to master mechanical engineering domain. They are simply required to push buttons, handle pedals and steering mechanisms whose functionality is described by natural language.

The contribution of the three major components of the domain-oriented ontology-based problem solving methodology to the problem solving process can be illustrated by Spiral project[PMJ+05] as follows:

- Domain ontology provides the framework for problem statement. For examples, Fourier transforms can be seen as an ontology for signal processing.
- The use of RDF-based languages (like OWL [MvH]) provides the mechanism that allow the identification of computer artifacts that implement stand-alone components of the solution algorithms. For example, programs implementing fast Fourier transforms can be seen as stand-alone components of algorithms solving digital signal processing, DSP, problems.
- The architecture description language allows the user to express her problems and solution algorithms using ontology vocabulary. For example, signal processing language (SPL) and the Spiral code generator maps SPL expressions into optimized code for given computing platforms.

The interpreter of SPL envisioned in this proposal would, however, map SPL expressions into processes performing the algorithms they represent.

Feasibility of a domain-oriented problem solving methodology that would simplify computer usage is now ensured by the maturation of the software technology generated by the huge amount of computer applications. Domain analysis is ripening as the ontology engineering [FSCB04, HM01, GM03, Miz, OLW03, MvH, PIS, SLCG99, SM03]; domain-oriented algorithm development and integration continues to advance with computer application; ontology development tools [Den, GP02, IP, HKR+] allow software architecture description languages to manipulate domain-abstractions rather than machine representations. Consequently, our task with this project is to synthesize all these achievements and integrate them into a domain-oriented ontology-based problem solving environment. This synthesis is an open-ended endeavor which can start with very simple and very well understood problem-domain (such as program development using a particular programming language, optimization methods using linear-programming, or any other fragment of a well understood domain of applications). The main virtue of such systems is that achievements performed by one iteration are generators of further achievements that enlarge and deepen the problem domain itself. Hence, it is almost obvious that such a problem-solving system mimics the teaching/learning aspect of the process of student-instruction in any kind of field of interest. The only new aspect we are preaching here concerns the effectivity and efficiency of the instruction process.

[1] We use the term *ontology* in this proposal in the sense defined by Smith[Smi] as *a classification of entities defined by a problem-domain's vocabulary and the canonical formulation of its theories*

Since the computer became a problem solving tool in any problem domain of human endeavor, we should be able to teach problem-solving using a computer within a given problem domain independent of the knowledge needed to understand computer science, exactly as a driving-instructor can teach car-driving independent of mechanical engineering. Car driving became ubiquitous and thus teaching car driving became a topic of general education; computer usage itself became (almost) ubiquitous, thus we should be able to move teaching computer usage outside of computer science, into the field of problem domain. This project promises a small-step toward this endeavor. But, as observed above, the virtue of this step is that it has the potential to create larger steps, i.e., it is meant to lead to the ubiquity of domain-oriented computer usage. While by ubiquity physical and logical computer will not disappear [SN05], computer existence will pervade unobserved, as many other human tools.

The advance achieved by formal methods in program-development [Cor] and software architectures [GMW02] have already created a methodology for domain-ontology development and this is demonstrated in various problem domains, such as medicine [HKR$^+$]. On the other hand, the progress realized toward automatic-program generation in particular domains such as compiler construction [Joh75, JL78, Hor88, MN04, Rus, Rus02, Rus03, PMJ$^+$05], and in general as discussed in [CE00], provides us with the first iteration of the vocabulary that is needed for the development of a domain-ontology. This vocabulary contains four kind of elements:

1. Specifications, which are mechanisms used in the domain of interest to express (formally) domain-problems. Examples are BNF rules in compiler construction, equations in mathematics, Ven-diagrams on optimization and scheduling, etc. These represent the basic elements of the language used to teach problem solving in the domain of interest.

2. Components, which represent stand-alone algorithms characterizing a domain of interest. Professionals of the domain can easily identify such algorithms.

3. Tools, which are universal algorithms that can map specifications into components. Examples are the mapping of BNF notations into automata that recognize the languages generated by such notations, or mapping of linear systems of equations into diagonal forms used by various solvers, or mapping of conditional probabilities into Bayes's rules applied in statistics.

4. Filters for component composition and interaction, which is achieved by I/O component standardization within the XML framework [Cona]. That is, (1) each component is specified by an $XML(input) \rightarrow Component \rightarrow XML(output)$ pattern; (2) component composition is performed by XSLT-filters, $XML(output) \stackrel{XSLT}{\longrightarrow} XML(input)$; and (3) component interaction is accomplished by appropriate stand-alone message and resource sharing systems.

The Resource Description Framework, RDF, [KC] provides the elements of the domain vocabulary with semantic expressions consisting of computer user independent implementation artifacts. For example, the term linear-system may have as numerical-analysis semantics a pattern describing the file-structure of a file representing a system of linear equations; LU-decomposition may have as numerical-analysis semantics a C-language program implementing an algorithm that maps a system of linear equations into a diagonal representation. Examples from any problem-domain abound because this is precisely what happens with the relationship between natural language terms and brain: natural language terms have as semantics their interpretation by the brain. The consequence is that computer users can manipulate the vocabulary of their domain of expertise as high-level abstractions independent of the machine representations that implement them. That is, domain-ontology ensures that computer users manipulate high-level computing abstractions rather than machine representations of data and operations, according to the logic of their problem, rather

than following the machine-logic of program execution [Rus03]. In addition, computer professionals in charge of developing problem-solving tools may be provided with architecture description markup-languages (ADML) [Gro99] whose semantics are domain-ontologies, thus providing mechanisms for application-integration within the XML type of languages. The interpreter of this language may use RDF representations of the domain-ontology to map problem solving architectures into computer processes performing the process of problems solving as specified by the computer user. The best illustration of this interpreter is provided by the Unix `make` system that maps a makefile that describes the process of a program generation by separate compilation of various components into the program thus described. The difference is that this interpreter will have to have access to Internet, to locate the semantics of terms used in architectural expressions, and to perform any kind of stand-alone programs, not just translators. This creates a relationship between an ADML expression and the process performing the computation described by this expression similar to the relationship between a Virtual Machine Monitor, VMM, [PPTH72, Gol74, PG74, FDF05] and the operating systems running under its control.

The above discussion implies that computer users need to be provided with computing environments populated by specification mechanisms, tools, and stand-alone software components. Specification mechanisms allow computer users to model their problems within a formal framework; tools are used to generate correct software components from correct specifications; stand-alone software components are algorithms characteristic to the problem domain that operate on data and function structures that may be generated by tools from specifications. For example, a PSE dedicated to solving systems of linear equations may consist of:

- **Specification:** $\sum_{i=1}^{n} a_{ji}x_i = b_j, j = 1, 2 \ldots, m$ where a_{ij} and b_j are real numbers and x_i are unknowns.
- **Tools:**
  1. Random number generators that generate n, m, a_{ij} and b_j for experimentation
  2. Input programs that map systems $\sum_{i=1}^{n} a_{ji}x_i = b_j, j = 1, 2 \ldots, m$ into internal files
  3. LU decomposition tools that map $\sum_{i=1}^{n} a_{ji}x_i = b_j, j = 1, 2 \ldots, m$ into triangular forms
- **Components:** LU decomposition solvers, say `LUsolver`

An architecture description language dedicated to the problem domain of *solving systems of linear equations* should be used by teachers and students studying this domain of expertise to express the architecture of the software systems that solve classes of problems in terms of the stand-alone components available within the PSE. For example, a program in the above environment would be:

$$\texttt{Solve } \sum_{i=1}^{n} a_{ji}x_i = b_j, j = 1, 2 \ldots, m \texttt{ by LU decomposition}$$

Interpreters would map these architectural expressions into processes solving the problems whose solution algorithm they represent. For example, an interpreter would map this expression into:

$$\texttt{LUsolver(LUdecomp(Input(}\sum_{i=1}^{n} a_{ji}x_i = b_j, j = 1, 2 \ldots, m\texttt{)))}$$

Note, the code performed by the machine while the interpreter handles `Input, LUdecomp, LUsolver` may reside in libraries that are accessible to the interpreter and may be written in any programming language implemented on the machine, because the processes performed by these codes are composed, not the code itself.

1.2 Objectives of this proposal

For the vision described above to happen, problem domains need to be identified, their ontologies need to be specified in terms of software artifacts that implement their vocabularies, and their solution algorithms need be expressed in terms of more primitive solution algorithms. That is, for a

given class of problems, solution algorithms need to manipulate processes represented by the more primitive solution algorithms rather than the language expression representing these processes. This is similar to mathematical thinking where mathematicians manipulate the meaning of the symbols they use by the rules provided by the meaning of these symbols, independent of the form of these symbols. That is, computers should be provided with application development tools that manipulate processes represented by software artifacts rather than the code expressing these software artifacts. Such application development tools are similar to control language interpreters (such as a shell in a Unix environment) which manipulate processes that perform computations expressed by problem solving algorithms rather than manipulating programming language expressions of these computations. Hence, the objectives of this proposal are:

1. Develop the methodology for application-domain structuring using ontology engineering tools and design software architecture description languages (ADL) that can capture problem-solving process as ADL expressions using domain-ontology;

2. Implement ADL interpreters that integrate computing abstractions (used in ADL expressions) and generate computing processes (performing the algorithms expressed by ADL expressions);

3. Use this methodology to conduct problem solving experiments in such diverse problem domains as Internet Agents, Geography, and Hydrology; integrate this problem solving methodology within the computing education by developing and teaching hands-on-experience courses.

First objective of this proposal will be achieved by creating experimental ontologies of problem domains implied in this proposal. For this purpose we will start by a short course on *hands-on ontology development* using Protègè model for domain ontology development [HKR$^+$], carried on on the model of program development using specifications, tools, components and XSLT filters given in TICS system [RKS$^+$, Rus03] and following Specware methodology [Cor]. This course will then be reproduced and extended in all the domains of interest, creating mini-ontologies of these domains while experimenting with this problems solving model. The second objective of the proposal will be achieved by identifying specific problem-models to be solved within the framework created by the first objective. This will be a continuous activity carried out by the participants to this projects while building up on previous experience. The third objective will be achieved by creating hands-on teaching experiments where instructors and students share ontology development tools to develop the ontology of the instruction domain and to express problems and solution using appropriate ADML. The mapping of ADML expressions into programs will then be carried out, first manually, as part of the teaching/learning process, and then developing appropriates interpreters to automate these mappings.

The major challenge of this project results from the recognition that the current approach for software system design hides systems complexity by encapsulating in the system design both the system architecture (the structure) and system function (the process). Hence, we build our approach for computer application development by separating the system architecture from the system function. The agents performing computations in general and Web Services [Hen01] in particular, are in the activity we propose, processes defined as tuples $< Agent, Expression, Status >$ where *Agent* is the processor that performs, *Expression* is the expression of the computation performed by the Agent, and *Status* is the status of the computation process. Contrasting this approach with the recent experiments carried out by W3C we notice that while W3C is concerned with data manipulation on the Internet, we are concerned with process manipulation by software systems where these processes solve the problems for which they were designed. In addition, while the infrastructure of the Semantic Web is developed by creating a framework for representing information in the Web

[KC], and the creation of a Web Ontology Language OWL[MvH], that supports reasoning about this information [CvHH$^+$], we are creating an infrastructure for process manipulation by separating the two components of a software system: the architecture and the function. Our objective is to develop a problem solving methodology with a computer that allows users to manipulate symbols representing system architectures, while system software (i.e., the tools populating the problem solving environment) manipulate processes representing the functions of the symbols manipulated by computer users. Therefore we develop problem domain ontologies taking as semantics of ontological terms stand-alone software artifacts, and implement tools to reason about software systems using these ontological terms, employing the set theoretical model proposed by W3C [Conb].

1.3 A process-based application development system (PADS)

Current software technology abounds in application development systems. An Internet search-tool (such as Google) finds hundreds of such systems, among which we notice [sIC, FOR, iL, Inc, Groa, Int, Grob]. All of these systems present methodologies for program development, customized to particular computer installations. There is nothing pertaining to problem-domain manipulation in these systems. The more academic based application development systems such as, CORBA developed by Object Management Group [Grob], GenVoca[BCRT02, BSR03], model-based systems [SK97], and PSE based[HRGB00] build on the same objective: *program development for a particular computing environment using a particular pattern of program generation.* The problem domain is usually approached by such systems through the mechanisms of *component-reuse* [Big98a, Big98b, Nei96, Nei98, Nei99].

The goal of the process-based model for application development system, PADS, is the creation of a domain-oriented methodology for problem solving, based on the separation of the architecture of a software system from the function performed by that system. System architecture is described by an Architecture Description Markup Language, ADML, while the system function continues to be described by any higher or lower level programming language, as currently is the case. The ADML used by a computer user is problem-domain oriented and is built on top of a vocabulary that represents the problem domain ontology. Each ontological term used by the ADML is associated with one or more stand-alone software artifacts implementing its function.

A problem solver uses a problem-domain oriented ADML to express the architecture of the system that solves her problem. An interpreter analyzes this architectural expression, locates the functions implementing various components of the architecture, creates the processes performed by these components, and composes these processes into the process represented by the architectural expression, thus solving the problem.

We use Acme [GMW02] as the model of architecture description language, ADL, and ADML [Gro99] as the framework for the ADL customization to the problem domain. The mini-ontologies we intend to create would have as basic properties that their vocabularies contain domain-characteristic specifications, tools, components, and XSLT filters provided with programming language semantics. This will allow us to carry out, first by hand, and then automatic, the mapping of the architectural expressions into computation processes. We will use our own problem solving environment provided by the TICS system, Technology for Implementing Computer Software, to design and implement problem-domain oriented ADLs and their interpreters. TICS is a framework for language design, implementation, and use. Hence, we will accomplish the objective of creating the process model for application development system, PADS, by the following sequence of steps:

1. Expand our current problem solving environment provided by the TICS system to support domain-oriented component-based PSE-s by implementing tools for problem domain ontology development and reasoning, which will be provided as teaching and learning tools.

2. Develop the domain-oriented software architecture description language SELScript provided with process control-flow based on ADML and TICS. SELScript will use the vocabulary of the problem-domain ontology to express system architectures while the code performed by the architectural components will be stored in software libraries.

3. Develop interpreters that map SELScript architectural expressions into processes performing the algorithms encapsulated in these expressions; these interpreters will use software libraries to generate processes composing the process expressed by SELScript expressions.

4. Provide TICS, SELScript, and SELScript interpreters as research, teaching, and learning tools to be used in the process of integrating research and teaching.

1.4 Integrating research and teaching

A teaching/learning approach that supports the process model for application development system, PADS, has been developed by the PI as the *hands-on-machine* approach. For this to succeed, students need unrestricted and unlimited access to a powerful computer along with software tools that can simulate and carry out the activities presented in the teaching process. Using this approach students create programming environments dedicated to their fields of interest at the beginning of the class. These environments are then improved and extended by hands on machine and problem. At the end of the teaching period these environments become major tools for solving target classes of problems in the domain of interest.

Among many other things, student contributions to education through the development and use of educational tools provide a beneficial feedback on the software technology itself. With the TICS project, the Department of Computer Science at The University of Iowa, has a long history of achievements in this area. The TICS research group consists mainly of students who used our parallel machines during their classes following the *hands-on-machine* approach, and succeeded in designing and implementing teaching tools that allowed us to successfully use this approach for various classes across several generations of students. Thus, the TICS research group has created a problem solving environment populated by specification rules, tools, components, and filters that allow students to develop processes that solve their problems using subprocesses that represent solutions to subproblems provided by components available in TICS environment. This methodology was successfully applied in such courses as Parallel Programming, Genetic Algorithms, Compiler Construction, and System Software. Currently, we are expanding the area of such courses by adding Internet Processing Environments and System Architecture Specification Languages. That is, the research aspect of this project concerns the integration of our experience with the *hands-on machine* approach with the research on software system design, implementation and use.

1.5 Intellectual merit of the proposed activity

Computing artifacts handled in this proposal are abstractions characteristic to the problem domain, represented by tuples $\langle syntax, semantics, ontology \rangle$ where *syntax* is the architectural expression handled by the user, *semantics* are the universal resource identifiers, URIs, of the stand-alone software artifacts implementing the tuple, and *ontology* are the URIs of the terms in the ontologies where they belong. This representation allows us to see the computing objects we manipulate three-ways: architectural, as blue prints represented by syntax; machine operational, represented by semantics URIs; and human logical, represented by ontological URIs. The separation of architecture (structure) from the function (the code implementing the tuple) and from the logic (ontology of the universe of discourse) characterizes all other engineering disciplines where objects manipulated are

similar representations, $\langle blue - print, sensorial, logic \rangle$. Therefore, we expect that the computing artifacts we manipulate will advance knowledge on software engineering across all field of endeavor where computers are used as problem solving tools. Specifically, this provides a logic foundation of the component-based domain-oriented software development approach by blue-print composition at the architecture level, process composition at the implementation levels, and logical proof at the ontological level. Consequently, this approach allows us to control software complexity by reusing expertise encapsulated in such artifacts. In addition, blue-print composition supports the advancement of natural language as a machine-communication tool and opens many possibilities for reproducing software artifacts and thus teaching them by experimentation rather than by textbook. The experience we gained with the TICS system is a guarantee of success.

1.6 Broader impacts of the proposed activity

Application development by computer process composition is the natural way for computation distribution in computer networks. The process management system we discuss here has the potential to embed PSEs within computer networks by locating and sharing resources. PSEs of a domain ontology can be used as the semantics for network clusters thus creating the mathematical infrastructure for reasoning about the computation activity performed by a computer network. Therefore this project has the potential to provided the theoretical foundation for semantic-search, for grid computing, and for network design. In addition, this proposal presents a new methodology to foster integration of research and teaching. The benefits of this integration go beyond the academic institution promoting it. Following the research, teaching, and learning approach discussed in this proposal students develop their own problem solving environments during their learning process. These environments are then further ported into the students' work-areas in the institutions where they are hired. Our previous experience shows that students' PSEs grow into problem solving environments in the respective institutions. This ensures an evolutionary process of discovery and understanding fostered by the process of problem solving itself. Moreover, since the process is grass-roots oriented it provides an equal opportunity for advancement.

2 Steps toward the new problem solving methodology

The four step methodology followed by a human while solving a problem, as explained by Polya [Pol73], implies: (1) understand the problem, (2) make a plan to develop a solution, (3) carry out the plan, and (4) look back at the completed solution. This methodology is not much different when software systems are used as tools for problem solving [RR95]. It still requires a user to understand the problem. But since software systems are used as tools to *automate problem solving*, the issue now is the development of a methodology for the generation of such tools. Software tools are, however, among the most complex artifacts produced by humans so far and their complexity induces more complexity in the methodology of their automatic generation. The current experience with software tool generation pertains mainly to the generation of software tools for program generation and execution not for problem manipulation (i.e, application development). Therefore we are still assisting to an expandable gap between applications and tools supporting computer applications. Hence, since there is little experience with tools for computer application generation, the intuition of a human-oriented methodology for problem solving with such tools needs to be developed. The work we currently perform may be seen as an experimental framework for the development of *software tool intuition*, which adds the *objective* along with the *abstraction* and *reflection*, as human aspects of the software engineering teaching and learning, preached by others [HT05].

2.1 Historical perspective

The term "problem solving environment", PSE, was coined early in computer history, and is illustrated by very simple PSEs developed as libraries of functions associated with various software systems. Examples of such software systems are operating systems and compilers. This idea has emerged again and the new meaning of this term has become "a computer system that provides all necessary computational facilities to solve a target class of problems" [GHR94, HRGB00]. The meaning of the PSEs illustrated by the libraries of functions mentioned above is no different. The target classes of problems in these cases are the management of program execution by operating systems and the management of program development by compilers. In other words, in building our experimental framework for the new problem solving methodology we observe that in the early PSEs as well as in their new instantiations, a PSE refers to a problem domain identified by: (1) a collection of specification mechanisms, (2) tools that operate on specification mechanisms mapping them into data and operations appropriate for components, (3) components that solve classes of problems by universal algorithms characteristic to the problem-domain, and (4) filters used as mechanisms for component composition. There is a dynamic relationship between tools, components, and filters where tools and filters may be used as components and components and filters may be used as tools. The net effect is a hierarchical methodology where problem solving algorithms are incrementally and interactively developed in terms of more primitive algorithms which are previously implemented. This is magnificently illustrated by the history of operating systems and compilers and is now being illustrated in other problem domains as well. Examples of such PSE-s in computational science [GHR94, HRGB00] are: PSEs for partial differential equations, PSEs for linear system analysis, PSEs for scientific computing and visualization, PSEs for interactive simulation of ecosystems, PSEs for network computing. All of these PSEs are collections of specifications, tools, components, and filters. However, these PSEs also have drawbacks which make them more complex than necessary and thus difficult to learn, use, and reproduce. Among these drawbacks we mention:

1. These PSEs are based on conventional programming language constructs, not on the natural language understanding of the specifics of the problems they solve.

2. The final product is a program on a given platform that is composed of components previously developed, perhaps on different platforms. Therefore, the usage of these systems requires users to master the programming objects they use rather than the problem-domain they approach.

3. There is no clear methodology for teaching, learning, and reproducing these systems.

Our idea is to embed such PSEs as vocabularies in the domain ontology and use architecture description languages to specify the system software that solve a class of problems. The appropriate interpreters, that map software architectures into processes that perform the functions the user desires to attach to such architectures, generate computation processes that solve problems. No programming as usual is implied. Consequently, while constructing a system from components, processes performed by the system components are composed, not the code that represents these processes. Since these processes are abstractions that represent problem domains, problem solving is based on domain-expertise not on the peculiarities of the computation platform. This is illustrated by the control language of a time-sharing system and by the extension of the TICS system reported in [Rus03].

2.2 Hands-on experiments

The phenomenon, characterized by researchers as ubiquitous computing [Wei91, LY02], raises new teaching and learning challenges. Traditional textbook-centered teaching methodologies are inade-

quate for educating students about concepts behind the relentless development of new computer-based "gadgets" such as PDAs, GPS receivers, and cell phones. The time needed to develop a textbook to teach the design, implementation, and usage of new computing gadgets can be longer than the time needed to develop even newer gadgets, and therefore, *a textbook may become obsolete before even being printed*. This situation leads to the requirement of a new educational methodology based on the availability of computing tools that are able to represent and implement the concepts and methods presented in the teaching process. With such tools, students have the hands-on means to repeatedly simulate and experiment with phenomena presented in the classroom. This phenomenon has been the object of the PI's observation during the last 20 years of teaching operating systems, compiler construction, system software, parallel programming, algorithm design, implementation, and use, and various computer applications, in the Computer Science Department at The University of Iowa. Our conclusion is that a major hurdle in teaching fast-changing computing technologies is the lack of tools that facilitate the *transformation of repetition into intuition and of intuition into knowledge*. This can be alleviated by the *hands-on-experience* approach to teaching, based on computer simulation, that consists of:

1. Unrestricted access to a computer with capabilities appropriate for the particular subject matter.

2. Software tools that can simulate various aspects of the teaching and learning process. In some cases, these tools can be generated by students during their own learning processes.

3. Hands-on instruction, in which the course material is illustrated using appropriate tools installed on the computer, combined with hands-on learning, in which students use the computer and installed software tools to create and experiment with new tools that implement course concepts.

4. Creation, by students, of personal portfolios (which are customized personal PSEs) based on their hands-on educational experiences. The software artifacts and tools created in hands-on courses have long-term value, in both later classes and ultimately, their professional careers.

By necessity, but perhaps not always consciously, many computer science instructors employ hands-on teaching methods. The PI first consciously used this approach in 1987 during the development of the course 22C:132, Parallel Programming, with the specific goal of developing student intuition for parallel computations. The relevant computer available at that time was an Encore parallel processor provided to us to experiment with parallel program development. The tools available for this task were a version of the Unix-V operating system and the usual programming languages and their compilers. The most important was the C language because it provides the programmer with a process creation capability during program development. Thus, process manipulation by students becomes feasible, and allows students to develop the intuition of a process abstraction that represents a schedulable and composable unit of computation that interacts with other processes.

2.3 Evolving hands-on experience approach

In 1992, the Committee on Physical, Mathematical, and Engineering Sciences of the U.S. Federal Coordinating Council for Science, Engineering, and Technology, assessed the major problems faced by new developments in US science and technology, and published the report *Grand Challenges: High Performance Computing and Communications* [RP]. This report identifies parallel computing as the major tool to solve grand challenge problems. This situation led to the national requirement to create "a culture" of parallel computing. Our experience with the parallel programming course

allowed the Department of Computer Science to respond quickly to this requirement by establishing a consortium for student education in high performance computing, consisted of four Iowa education institutions: The University of Iowa and three outstanding private colleges, Cornell, Graceland, and Luther. The goal of this consortium was to enable education in parallel computing by the *hands-on-machine* approach at both graduate and undergraduate levels. Consequently, a proposal to acquire an appropriate computer was submitted to NSF by Prof. T. Rus (Computer Science), PI and CoPIs Prof. F. Potra (Computer Science), Dr. J. Brown (Weeg Computer Center), Prof. J. Jones (Graceland College), Prof. T. deLaubenfels (Cornell College), and Prof. W. Will (Luther College). This proposal was funded by the NSF grant DUE–9551183 ($215,369.60) with a match from The University of Iowa. Thus, in 1996, the by-then-obsolete Encore computer was replaced by an SGI Power Challenge equipped with 16 R10000 processors. To achieve our goal we developed new courses, reshaped older courses in this area, and today we can report that hundreds of Iowa students benefited from the use of this machine. In addition, we generated new research activities on high performance computing and enhanced our curriculum in both computer science and applied mathematics.

We are now facing a new grand challenge problem, which may be characterized as the *source of all grand challenge problems*. It is the challenge to develop a problem solving methodology that is able to simplify the use of current and future computer technologies to solve other grand challenge problems, and to control the complexity of software system development, installation, and use. This problem, once again requires us to unify forces on the University of Iowa campus by creating an Interdisciplinary Computing Laboratory to play the role of a test-bed for the problem solving methodology we are currently creating. At the basis of this Laboratory sits our experience with problem solving environments, as reported in the literature mentioned above, and the results we obtained with the consortium for the development of a critical mass of parallel computer users. The Interdisciplinary Computing Laboratory will be established by joining forces of major users of high performance computer technology on campus, namely:

1. Marc Armstrong, Department of Geography, developing mathematical models of geographic domains and using high performance computing to solve such problems.

2. George Constantinescu, Department of Civil and Environmental Engineering, using mathematical modeling for the development of tools to predict hydrological processes and large scale parallel computing to predict multi-scale environmental processes.

3. Teodor Rus, Department of Computer Science, developing models of domain-ontology engineering, system architecture description languages, and interpreters that map architectural expressions into processes performing computations they represent.

Experiments developed by students attending the courses on parallel programming over the last 15 years cover all these areas and illustrate the new problem solving methodology we are developing with PSEs provided as final projects for the parallel programming class. Therefore, we expect that the funding of this project will expand further the results obtained so far, leading to both improved education in science, technology, and engineering through adaptation and implementation of the educational practices that we have developed before, and to the concrete use of the new problem solving methodology. Our belief is based on the fact that the hands-on approach to education in high-performance computing gained in popularity in the University of Iowa over the last 10 years at both graduate and undergraduate levels, thus the necessity of a new problem solving methodology became obvious.

2.4 Reusing educational experience

The SGI Power Challenge, purchased in 1996, is no longer in use. Meanwhile, high performance computing has expanded substantially, and greater challenges have been created for student education in this area. Therefore, replacement of the SGI Power Challenge is important to the continued health and further enrichment of the high performance computation curriculum in two key ways:

1. It will directly support existing and planned courses on high performance computing education. We will expand our courses on parallel programming and parallel algorithms with new application-oriented courses in all areas of research mentioned above.

2. It will facilitate the building of a computing education environment based on application-domain structuring using ontology engineering tools and software specifications as architectural expressions, using machine independent system architecture description languages.

To achieve these goals we need to acquire a new parallel machine as a tool to facilitate expanding the *hands-on-machine* approach to a larger educational context. This context is provided by new developments in high performance computing that are mainly generated by interdisciplinary research. The new machine will be used by the Interdisciplinary Computing Laboratory as the main computing facility to experiment with the new problem solving methodology we develop. It will also be offered to all instructors who would like to teach their students high performance computing using the hands-on approach and experiment with the domain-oriented ontology-based problem solving methodology. Based on our past experience, we expect an average enrollment of 30 students per class taught by one instructor, in each of these classes. Hence, approximately 300 students per year would benefit from the services offered by this machine and will experiment with the new problem solving methodology we create.

Adding the creation of an education environment based on ontology engineering tools and software architecture description languages to our goals is a natural consequence of our experience which shows that the biggest barrier to machine-independent thinking in software engineering is the deep intertwining in today's software of two things of a different nature: system architecture and system function. Due to new accomplishments in software development achieved by us and by others [All02, LS02, RKS$^+$] and problem-domain structuring by ontology engineering tools [Gru93b, Gru93a, OLW03, HM01, GM03, Wel03, Miz] the *hands-on experience* approach, which was successfully used for teaching parallel programming, now can be successfully used to perform the first step toward the domain-oriented ontology-based problem solving methodology. This will allow teachers and students to develop the intuition of their software systems as abstractions independent of the functions they perform. Moreover, in the long term, this provides a foundation for the use of natural language as a man-machine communication tool.

3 Plan of Work

Our plan of work evolves on two parallel tracks: on one track we will pursue the implementation of the process-based model for the application development system (PADS) and on the other track, all PSEs involved will create in parallel and in parallel with the first track the ontologies of their problem domains and will develop models of problem solving architectures. These models of problem solving architectures will then be used as tests for the PADS and will be evaluated in both graduate and undergraduate teaching.

Track 1: Process-based Application Development System, PADS

1. We will first port the tools developed so far using the Encore and SGI Power Challenge machines onto the new machine. This activity will be carried out by Prof. Rus and one graduate student during the Spring semester 2006.

2. In parallel with the activity described in (1) above we will start using the new machine during the Spring semester 2006. The PI will teach a course on tractable description logic (DL) ontology specification languages using OWL DL illustrated by a domain-ontology engineering tool such as Protègè, following the model presented in [HKR$^+$]. Other ontology development tools [Den, GP02, IP] will also be experimented.

3. We will finish the implementation of the SELScript during 2007. Software architectures and ontologies already developed in each of the PSE of interest will be used to test this implementation.

4. Starting in Spring 2008 we will use the new machine to run large computational jobs supplied by other PSEs involved in this project as well as ontological reasoning tools checking the consistency of these software architectures.

Of course, in parallel with this activity we will use SELScript and its interpreter to generate the tools required by the other PSE involved in this project.

Track 2: Domain Ontologies and System Architectures

Internet PSE Domain:

1. During the year 2006 Professor Rus helped by an RA will identify and implement the Semantic Web vocabulary of interest for this project. The goal of this activity is the development of an architecture specification language suitable to express the architecture of new Internet agents in terms of the agents already available. This vocabulary will contain the specification mechanisms, tools, components and XSLT-filters, available for agents specification and implementation.

2. During 2007–2008 we will generate the architectural expressions of the Internet agents required by the information exchange between the ontologies created by other problem domains involved in this project. We will design and implement ontological tools that will allow us to check the validity of the operations performed by these agents, and will initiate the development of Internet semantic search tools.

Geography PSE Domain:

1. In Spring 2006 Professor Armstrong will initiate, with the assistance of a research assistant, the specification of an ontological framework for geospatial information and its use in a particular context of the interpolation of data values from point observations taken from different sources to a regular grid of values. This will require the specification of data commonalities to assess congruence among data sets [MST99], [SM03]. The correctness of this specification will be proved by appropriate ontological tools developed on track one in Fall 2006.

2. Simple matters such as translating between a postal address and its coordinate reference will be specified by appropriate XSLT filters in parallel with the ontological domains. We plan to have this task completed by Spring 2007.

3. During 2007-2008, a set of generic processes will be identified and architectures of software systems solving various geographic problems will be expressed in terms of these processes. These include, for example, generic neighborhood specifications (e.g., k-nearest neighbors of a point), generic distance calculations in both Euclidean (projected) and spherical spaces, and different ways of making interpolation calculations (global and local methods). This work will be tested using SELScript and will be evaluated, first in a graduate-level seminar, in Fall 2007, and then in an advanced undergraduate class (Principles of Geographic Information Systems) in Spring 2008.

Hydrology PSE Domain:

1. In Spring 2006 Professor Constantinescu will initiate, with the assistance of a research assistant, the specification of an ontological framework for solving scalar transport equations in two and three dimensions. This ontology will be used in the particular context of predicting the fate and transport of pollutants in rivers and atmosphere.

2. In Spring 2007 the vocabulary of this ontology will be implemented by a set generic processes to discretize the connective and viscous operators in the transport equation using different levels of accuracy and different levels of numerical dissipation (e.g., central differencing with fourth order artificial dissipation versus upwind or upwind biased schemes of different order).

3. During 2007-2008 we will use the ontology developed above to design the architecture of software systems for solving various environmental process flow problems. These systems will be implemented using the PADS system developed on track 1.

3.1 Project evaluation

The evaluation of this project will be measured by the versatility of the PADS developed on track one of our plan, by the number and the quality of applications developed using the PADS within the track two of our plan, and by the number of students instructed by the *hands-on-machine* approach using the machine we seek support for. The number and the diversity of the domain ontologies embedded as vocabularies within the new TICS environment will be a good measure of the work we intend to perform. The PI and CoPIs will assess the progress toward stated project goals by making sure that the plan of the activity presented above will be carried out successfully. In addition, at the completion of this plan, during the year 2008 we will report our activities in a number of papers at national and international conferences, thus contributing to the dissemination of the teaching and research activity performed during the first two years.

We will organize two workshops, one in Fall semester 2007 and one in Spring semester 2008, where we will present in detail the experience accumulated by the PI and CoPIs with the process based methodology for problem solving with a computer. The educational results, obtained by us during this period using the *hands-on-machine* approach, will be demonstrated and shared with all interested. These workshops will discuss the merits and drawbacks of the hands-on approach with all those interested. We will use these discussions to generate detailed documentation of this approach showing its efficacy for teaching domain-oriented ontology-based problem solving methodology.

Project evaluation will also assess the impact of the publication of conference and journal papers, and production of PhD theses related to the work we plan to perform in this project.

4 Results from Prior NSF Support

Teodor Rus:

NSF award CCR 99-87466, $10,000, year 2000, PI Teodor Rus.

Project title: Organize the 8-th edition of the International Conference Algebraic Methodology And Software Technology, AMAST 2000.

PI: Teodor Rus

The major results are the volume 1816 of Lecture Notes in Computer Science that publishes the proceedings of AMAST 2000 and the Volume 291, Number 3 of Theoretical Computer Science Journal that publishes the best papers presented at this conference.

Marc P. Armstrong:

NSF SES-0345441, $22,311, year 2004, PI: Karr and Armstrong

Project title: Statistical Disclosure Limitation for Geospatial Image Data,

PIs: Karr and Armstrong, M.P., Duncan, G., and Sanil, A.

Award to National Institute for Statistical Sciences, sub-Award to University of Iowa.

References

[All02] editor Allen, P. Component development strategies. *The Monthly Newsletter from Cutter Information Corp. on Managing and Developing Component-Based Systems*, January 2002.

[Bax] I. Baxter. Dms (the design maintenance system). http://www.semdesigns.com.

[BCRT02] D. Batory, G. Chen, E. Robertson, and Wang T. Design wizards and visual programming environments for genvoca generators. *IEEE Transactions on Software Engineering*, 26(5):441–452, 2002.

[Big98a] T. Biggerstaff. A perspective of generative reuse. In *Annals of Software Engineering*. Springer–Verlag, 1998. Author's website: softwaregenerators.com.

[Big98b] T.J. Biggerstaff. Pattern matching for program generation: A user manual. Technical Report MSR-TR-98-55, Microsoft Research, Microsoft Corporation, December 1998. Author's website: softwaregenerators.com.

[BSR03] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step–wise refinement. In *International Conference on Software Engineering 2003 (ICSE J2003), Proceedings*, Portland, Oregon, May 2003, 2003.

[CE00] K. Czarnecki and U.W. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[Cona] The World Wide Web Consortium. Extensible markup language. http://www.w3.org/XM.

[Conb] The World Wide Web Consortium. W3c, technology and society domain – semantic web activity –. http://www.w3.org/2001/sw.

[Cor] Kestrel Development Corporation. Specware 4.0 tutorial 2002. Available at http://www.specware.org/documentation.

[CvHH+] D. Connolly, F. van Harmelen, I. Horrocks, D.L. McGuinness, P.F. Peter F. Patel-Schneider, and L.A Stein. *DAML+OIL Reference Description, W3C Note 18 December 2001*. Available at http://www.w3.org/TR/daml+oil-reference.

[Den] M. Denny. Ontology building: A survey of editing tools. Available at http://www.xml.com/pub/a/2002/11/06/ontologies.html.

[FDF05] R. Figueiredo, P.A. Dinda, and J. Fortes. Resource virtualization renaissance. *Computer*, 38(5):28–31, 2005.

[FOR] Inc. FORTH. Windows application development. http://www.forth.com/swiftforth/swiftforth-3.html.

[Fre04] P.A. Freeman. Science of design. *Computing Research News*, January 2004.

[FSCB04] J.M. Fielding, J. Simon, W. Ceusters, and Smith B. Ontological theory for ontological engineering: Biomedical systems information integration. In *Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning (KR2004)*, Whistler, BC, 2-5 June 2004.

[GHR94] E. Gallopoulos, E.N. Houstis, and J.R. Rice. Computer as thinker/doer: problem solving environments for computational science. *IEEE Computational Science and Engineering*, 2:11–23, 1994.

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

[GMW02] D. Garlan, R.T. Monroe, and D. Wile. *Acme: Architectural Description of Component-Based Systems*, chapter Part One, Section 3, pages 47–67. Cambridge University Press, 2002.

[Gol74] R.P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[GP02] et al. Gomez-Perez, A. A survey on ontology tools. ontoweb deliverable 1.3. Universidad Politecnia de Madridi, 2002.

[Groa] Symmetry Group. Sapphire application developmentsystem. http://www.symmetrygroup.com/.

[Grob] The Objecty Management Group. About the object management group. http://www.omg.org.

[Gro99] The Open Group. Adml: An xml based architecture description language. http://www.opengroup.org/architecture/togaf/bbs/9910wash/adml.pdf, 1999.

[Gru93a] T.R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. In *Proceedings, Padua Workshop on Formal Ontology*, 1993.

[Gru93b] T.R. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993.

[Hen01] J. Hendler. Agents and the semantic web. *IEEE Intelligent Systems*, pages 30–37, March-April 2001.

[HKR$^+$] M. Horridge, H. Knublauch, A. Rector, R. Stevens, and C. Wroe. A practical guide to building owl ontologies using the protègè-owl plugin and co-ode tools, edition 1.0. http://www.co-ode.org/resources/tutorial/ProtegeOWLTutorial.pdf.

[HM01] P. Hayes and C. Menzel. A semantics for knowledge interchange format. In *Proceedings IJCAI 2001*, Aug 6 2001. Available at http://reliant.teknowledge.com/IJCAI01/.

[Hor] P. Horn. Autonomic computing: Ibm's perspective on the state of the information technology. http://www.research.ibm.com/autonomic/manifesto/.

[Hor88] R.N. Horsopool. Ilair: An incremental generator of lalor(1) parsers. In *Lecture Notes in Computer Science 371*, pages 128–136. Springer-Verlag, 1988.

[HRGB00] E.N. Houstis, J.R. Rice, E. Gallopoulos, and R. Bramley, editors. *Enabling Technologies for Computational Sciences*. Kluwer Academic Publishers, 2000.

[HT05] O. Hazzan and J.E. Tomayko. Reflection and abstraction in learning software engineering's human aspects. *Computer*, 38(6):39–45, 2005.

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

[iL] iSculptor Limited. Sculptor, a portable application development.

[Inc] DataGlyphics Inc. Dataglyphics – application development. http://www.datag.com/.

[Int] Internosis. Notes2office system jumpstart. hhtp://www.internosis.com.

[IP] Maryland Information and Network Dynamics Lab Semantic Web Agents Project. Swoop - a hypermedia-based featherweight owl ontology editor. http://www.mindswap.org/2004/SWOOP/.

[JL78] S.C. Johnson and E. Lesk. Language development tools. *Bell System Technical Journal*, 57(6):2155–2175, 1978.

[Joh75] S.C. Johnson. Yacc – yet another compiler compiler. Computing science technical report 32, AT&T Bell Laboratories, Murray Yill, N.J., 1975.

[KC] G. Kline and J.J. Caroll. W3c, resource description framework (rdf): Concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/.

[LS02] G.T Leavens and M. Sitaraman, editors. *Foundations of Component-Based Systems*. Cambridge University Press, 2002.

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

[Miz] R. Mizoguchi. Ontology engineering environments. http://www.ei.sanken.osaka-u.ac.jp/pub/miz/HandBookMiz3.pdf.

[MN04] S. McPeak and G.C. Necula. Elkhound: A fast, practical glr parser generator. In *Lecture Notes in Computer Science 2985*, pages 73–88. Springer-Verlag, 2004.

[Mon01] R.T. Monroe. Capturing software architecture design with armani. Technical report, Carnegie Mellon University, Technical Report CMU-CS-163, 2001.

[MST99] D.M. Mark, B. Smith, and B. Tversky. Ontology and geographic objects: An empirical study of cognitive categorization. In *Spatial Information Theory: A Theoretical Basis for GIS*. Springer-Verlag, 1999.

[MvH] D.L. McGuinness and F. van Harmelen. *OWL Overview, OWL Web Ontology Language Overview. W3C Proposed Recommendation 15 December 2003*.

[Nei80] J.M.. Neighbors. *Software Construction using Components*. PhD thesis, University of California, Department of Information and Computer Science, Irvine, PhD thesis, 1980. Author's website: James.Neighbors@BayfrontTechnologies.com.

[Nei96] J.M. Neighbors. Finding reusable software components in large systems. In *Third Working Conference on Reverse Engineering*, pages 1–9. IEEE Press, 1996. Author's website: James.Neighbors@BayfrontTechnologies.com.

[Nei98] J.M. Neighbors. Domain analysis and generative implementations. In *Fifth International Conference on Software Reuse (ICSR5)*, pages 356–357, Victoria, Canada, June 1998. IEEE Press. Author's website: James.Neighbors@BayfrontTechnologies.com.

[Nei99] J.M. Neighbors. Transforming experiences. In *International Workshop on Software Transformation Systems, (STS'99)*, Los Angeles, May 1999, 1999. Author's website: James.Neighbors@BayfrontTechnologies.com.

[OLW03] L. Obrst, H. Liu, and R. Wray. Ontologies for corporate web applications. *AI Magazine*, 24(3):49–62, 2003.

[PG74] G.J. Popek and R.P. Goldberg. Formal reguirements for virtualizable third generation architectures. *Communications of the A.C.M.*, 17(7):412–421, 1974.

[PIS] L. Pouchard, N. Ivezic, and C. Schlenoff. Ontology engineering for distributed collaboration in manufacturing. http://www.mel.nist.gov/msidlibrary/doc/AISfinal2.pdf.

[PMJ+05] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2), 2005.

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

[PPTH72] R.P. Parmelee, T.I. Peterson, C.C. Tillman, and D.J. Hatfield. Virtual storage and virtual machine concepts. *IBM System Journal*, 11(2):99–130, 1972.

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

[RP] The FY 1992 US Research and Development Program. Grand challenges: High performance computing and communications. Published by Committee on Physical, Mathematical, and Engineering Sciences, The FY 1992 U.S. Research and Development Program.

[RR95] T. Rus and D. Rus. *System Software and Software Systems – Vol 1.* World Scientific, 1995.

[Rus] T. Rus. Ipc: Interactive lr tool parser construction. Lecture notes on compiler construction. Available at http://www.cs.uiowa.edu/ rus/Courses/Compiler/Notes/ipc.pdf.

[Rus02] T. Rus. U unified language processing methodology. *Theoretical Computer Science*, 281(1–2), 2002. A mosaic in honour of Maurice Nivat.

[Rus03] T. Rus. Domain-oriented component-based automatic program generation. In *VIP Forum on Interdisciplinary Computing, Proceedings*, Montenegro, October 5–11 2003. Available at http://www.cs.uiowa/edu/~rus/.

[sIC] Teta srl Internet Consultant. Application development system, stratos. http://www.stratos.org/.

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

[SLCG99] C. Schneloff, D. Libes, M. Ciocoiu, and M. Gruninger. Process specification language (psl): Results of the first pilot implementation. In *Proceedings of IMECE*, Nashville, Tennessee, USA, November 14–19 1999.

[SM03] B. Smith and D.M. Mark. Do mountains exist? toward an ontology of landforms. *Environment & Planning B: Planning and Design*, 30(3):411–427, 2003.

[Smi] B. Smith. Ontology and information systems. http://ontology.buffalo.edu/smith/articles/ontologies.htm

[SN05] N. Streitz and P. Nixon. The disappearing computer. *Communications of the ACM*, 48(3):33–71, March 2005.

[Wei91] M. Weiser. The computer for the 21st century. *Scientific American*, pages 94–104, September 1991.

[Wel03] C. Welty. Ontology research. *AI Magazine*, 24(3):11–12, 2003.