Spring 2013

An evolutional domain oriented approach to problem solving based on web service composition

Cuong Kien Bui
University of Iowa

AN EVOLUTIONAL DOMAIN ORIENTED APPROACH TO PROBLEM SOLVING BASED ON WEB SERVICE COMPOSITION

by

Cuong Kien Bui

An Abstract

Of a thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

May 2013

Thesis Supervisor: Professor Emeritus Teodor Rus

ABSTRACT

Computers are around us and integrated deeply in almost every aspect of human life. Computers are used to solve more and more types of problems in human lives. Software tools are designed to ease the process of integrating computers into our problem solving process.

To use computers to solve a problem with current software technology a computer user can either buy a software designed specifically for that problem or she needs to learn a general computer programming language to write computer programs to solve the problem. Even though, with computer software the user can solve that specific problem; it is still challenging to truly use that software as a tool to integrate the computer within the problem solving process. On the other hand, learning a computer language is not an easy task for most domain experts such as chemists, biologists, etc. The level of skills required for a domain expert to be able to translate a domain concept to a computer language concept is also high.

As an alternative, we want to create tools that enable problem solvers to express problem solving solutions in terms characteristic to their own domain and carry out problem solving processes in those terms. This thesis provides a contribution to the domain oriented software development and describes an implementation of this approach as a prototype system called DALSystem. In this approach, a problem domain is first formalized using a domain ontology, then the domain expert expresses her solution algorithm using the terms of that ontology. The expression of her solution algorithm is then translated to

the intermediate language of a domain dedicated virtual machine (DDVM) and is evaluated by an interpreter using the domain ontology. The solution algorithm can later be imported into the domain ontology thus expanding the problem domain with new concepts (action and data) in a process called Domain Ontology Evolution (DOE).

With this methodology, the DALSystem can execute algorithms whose expressions are conceptual, similar to the way the human brain would execute them. We illustrate this methodology using DALSystem in the domain of arithmetic.

Abstract Approved: _____
Thesis Supervisor

_____
Title and Department

_____
Date

AN EVOLUTIONAL DOMAIN ORIENTED APPROACH TO PROBLEM SOLVING BASED ON WEB SERVICE COMPOSITION

by

Cuong Kien Bui

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

May 2013

Thesis Supervisor: Professor Emeritus Teodor Rus

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Cuong Kien Bui

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the May 2013 graduation.

Thesis Committee: _____
Teodor Rus, Thesis Supervisor

_____
James Cremer

_____
Alice Davison

_____
David Eichmann

_____
Juan Pablo Hourcade

_____
Gregg Oden

ACKNOWLEDGEMENTS

I would like to thank first the members of my thesis committee: James Cremer, David Eichmann, Alice Davison, Gregg Oden, Juan Pablo Hourcade, and especially my advisor, Teodor Rus, who suggested the topic for this work and patiently provided enormous guidance throughout.

David Eichmann also deserves thanks for providing an innovative research environment and stimulating ideas about the applications of ontologies and language processing during his student meetings every week. I also would like to thank other students of Institute of Clinical and Translational Science: Todd Papke, Si-Chi Chin, Charisse Madlock-Brown, Ray Hylock, Brandyn Kusenda and Jimmy (James Schappet) for their support, they were interesting people to have around.

I would like to thank my parents, Quoc Kien Bui and The Thi Nguyen, for giving me the opportunities to do these things. I can't thank them enough, but at least I can say "Con cam on bo me that nhieu!" The next two very important people to me are my wife Hang Nguyen and my son Bao Bui. I would like to thank them for their love, support and encouragement me to finish this work. I am also pleased to acknowledge all the help of my sister during the last 4 months of this work. Thanks, guys.

Finally, I would like to thank the Department of Computer Science at the University of Iowa for their support, especially Sheryl Semler and Catherine Till. I also would like to thank Vietnam Education Foundation (VEF) for granting me the PhD fellowship.

ABSTRACT

Computers are around us and integrated deeply in almost every aspect of human life. Computers are used to solve more and more types of problems in human lives. Software tools are designed to ease the process of integrating computers into our problem solving process.

To use computers to solve a problem with current software technology a computer user can either buy a software designed specifically for that problem or she needs to learn a general computer programming language to write computer programs to solve the problem. Even though, with computer software the user can solve that specific problem; it is still challenging to truly use that software as a tool to integrate the computer within the problem solving process. On the other hand, learning a computer language is not an easy task for most domain experts such as chemists, biologists, etc. The level of skills required for a domain expert to be able to translate a domain concept to a computer language concept is also high.

As an alternative, we want to create tools that enable problem solvers to express problem solving solutions in terms characteristic to their own domain and carry out problem solving processes in those terms. This thesis provides a contribution to the domain oriented software development and describes an implementation of this approach as a prototype system called DALSystem. In this approach, a problem domain is first formalized using a domain ontology, then the domain expert expresses her solution algorithm using the terms of that ontology. The expression of her solution algorithm is then translated to

the intermediate language of a domain dedicated virtual machine (DDVM) and is evaluated by an interpreter using the domain ontology. The solution algorithm can later be imported into the domain ontology thus expanding the problem domain with new concepts (action and data) in a process called Domain Ontology Evolution (DOE).

With this methodology, the DALSystem can execute algorithms whose expressions are conceptual, similar to the way the human brain would execute them. We illustrate this methodology using DALSystem in the domain of arithmetic.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

LIST OF ALGORITHMS

Algorithm

LIST OF LISTINGS

Listing

CHAPTER 1
INTRODUCTION

The concept of “Liberating Computer User from Programming” first appeared in 2008 (Rus 2008). This concept does not mean that programming would disappear; rather, it means that while computer programming will be performed by computer programmers, computer use in a domain of application will be performed by a domain expert using a domain algorithmic language (DAL) (Rus 2013). This implies the development of software tools that allow the computer user to use the computer transparently during her problem solving process. In other words, the user can use the computer for solving problems without worrying about the computer platform they are running on and the computer language used to program the solution algorithm. Therefore, in the resulting software methodology, the computer is considered as a tool integrated into human problem solving process. This thesis provides an implementation of this idea for the domain of arithmetic.

1.1 Problem Solving Process on Computers

Originally, computers have not been developed as problem solving tools. Instead, they were invented by mathematicians and engineers as number crunching tools. Within the framework of original creators of computers, the computer use during problem solving process follows Polya’s (1945) problem solving methodology and consists of the following steps:

1. Formulate the problem;
2. Develop a solution algorithm;

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

Even though this approach of using the computer as a problem solving tool serves the original creators (physicists and mathematicians) well to some extent, it does require the computer user to understand computer architecture and functionality to be able to encode the algorithm into a program. This requirement turns out to be a huge obstacle for other domain experts such as chemists, biologists and engineers, to be able to use computers for their computations. Computer experts then try to diminish this difficulty by developing software tools like operating systems, programming languages, compilers and interpreters. The idea of these software tools is to raise machine language abstraction level towards the logical level of problem solving process. Therefore these software tools abstract away the thinking in terms of binary signal processing at the machine level. But the machine computation concepts which these tools use do not represent the concepts used by the domain experts during the human problem solving process. They represent concepts that belong to the computer architecture and functionality. Therefore, in order to use the computer during the problem solving process, the computer user needs to learn computer architecture and functionality as well as the new language provided by software tools. Consequently, this framework requires a higher level of professionalism from the computer user. As the number and the complexity of problems domains increases, the complexity of the software tools supporting the problem solving process by translation from problem domain language into software tools languages increases dramatically. As a consequence, this framework of fit-

ting the problem solving process within the computer increases the complexity of computer software to a level where it threatens to kill the computer technology itself (Horn 2001).

This thesis proposes a solution for the problem of integrating computers in the problem solving process by making the steps (1) and (3-4) of the computer-based problem solving process easier for computer users. In our approach, domain concepts are first organized in an ontology using domain characteristic terms. Those concepts are then associated with their computational meanings with some initial help from computer experts. Next, the solution algorithm in steps (3-4) can be written in these domain terms, while the algorithm can later be executed on a computer network by a virtual machine which searches the computational meanings of these domain terms in the domain ontology. This thesis also provides a working system to demonstrate our approach, called DALSystem.

1.2 Domain Modeling Using Ontologies

The first step in our approach to the problem solving process is to organize problem domain concepts in an ontology. Domain experts perform this step by recognizing concepts that characterize the problem domain. This step in problem formalization means that the problem solver defines problem concepts and methods in terms of well-understood concepts and methods. Using a mathematical saying, “one cannot expect to be able to solve a problem one does not understand". For example, in the domain of arithmetic such concepts are `number, integer, real, add, multiply, subtract, divide`, etc. For the domain of computational linguistics, such concepts are `word, phrase, sentence, category, parse tree, parse, stemmer`, etc. Those characteristic concepts

form the *pure domain ontology*. Methodologies and tools for constructing such ontologies can be found in (Welty & Guarino 2001, Gasevic, Djuric & Devedzic 2009). Figure 1.1 shows an example of how arithmetic domain concepts can be organized.

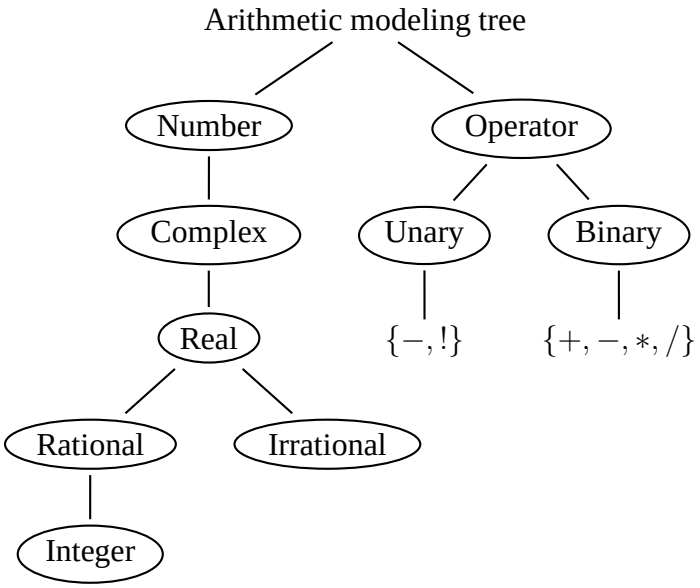

Figure 1.1: Arithmetic modeling tree

Our conjecture here is that solvable problems of any problem domain are expressible in terms of a finite number of well defined concepts. This is trivially true for the common sense problems raised by the usual real-life. A formal proof of this conjecture can actually be sought using decidability theory (Sipser 2006).

1.3 Domain Solution Algorithm and Web Execution

The next step in our approach is the collaboration between domain expert and computer expert to associate concepts in the pure domain ontology with their computational meaning implemented by web services or XML data types. I choose to use web services using industry standards such as SOAP (Box, Ehnebuske, Kakivaya, Layman, Mendelsohn, Nielsen, Thatte & Winer 2000), WSDL (Christensen, Curbera, Meredith & Weerawarana 2001), UDDI (Clement, Hately, von Riegen & Rogers 2004) as the implementation of computational meanings of domain concepts to make the system have a better impact on the community. In this step, the computer expert uses a meta-ontology called CEADOntology, which will be discussed later in Section 3.2, to associate each domain concept with its computational meaning as

- an execution agent if the domain concept is an action concept such as `add, multiply, subtract, divide,` etc.
- an XSD data type if the domain concept is a data concept such as `integer, real,` etc.

Each execution agent could be implemented by several service instances so that if one service instance is not available another one can take its place.

Finally, to support the domain expert in expressing her computation in domain terms, a language specific to that domain is created with the help of a computer expert. However, unlike other domain specific languages (DSL) where the meaning of each expression is fixed, this language only provides a general mechanism for logically composing meanings of domain terms. The concrete meanings of domain terms are specified by the

ontology. Therefore the meaning of each expression in this language is inherently dynamic, depending on the state of the ontology.

Problem solutions (algorithms) are then expressed in terms of concepts and operations characteristic to the domain. These expressions are actually valid expressions in the domain language of the problem solvers, which are understood by all domain experts because these expressions use only concepts familiar to the domain experts.

Solution algorithms to the problem solved this way can be stored in the domain ontology by tuples $(term, solution\ algorithm)$. This way the knowledge obtained by problem solving become new domain concepts that can be reused to solve other problems. This is the domain evolution process that can be iterated indefinitely. Thus, the user domain ontology will expand indefinitely during the process of the domain expert solving her problems.

1.4 System

To support our approach to the problem solving process, we identify the following software tools as needed:

1. Tools for domain specification using an ontology. Protege [1] is an excellent off-the-shelf tool for this purpose.
2. A virtual machine which operates on domain ontology to prepare data and make appropriate calls to web services implementing action concepts, called Domain Dedicated Virtual Machine (DDVM).

[1] Available at http://protege.stanford.edu/

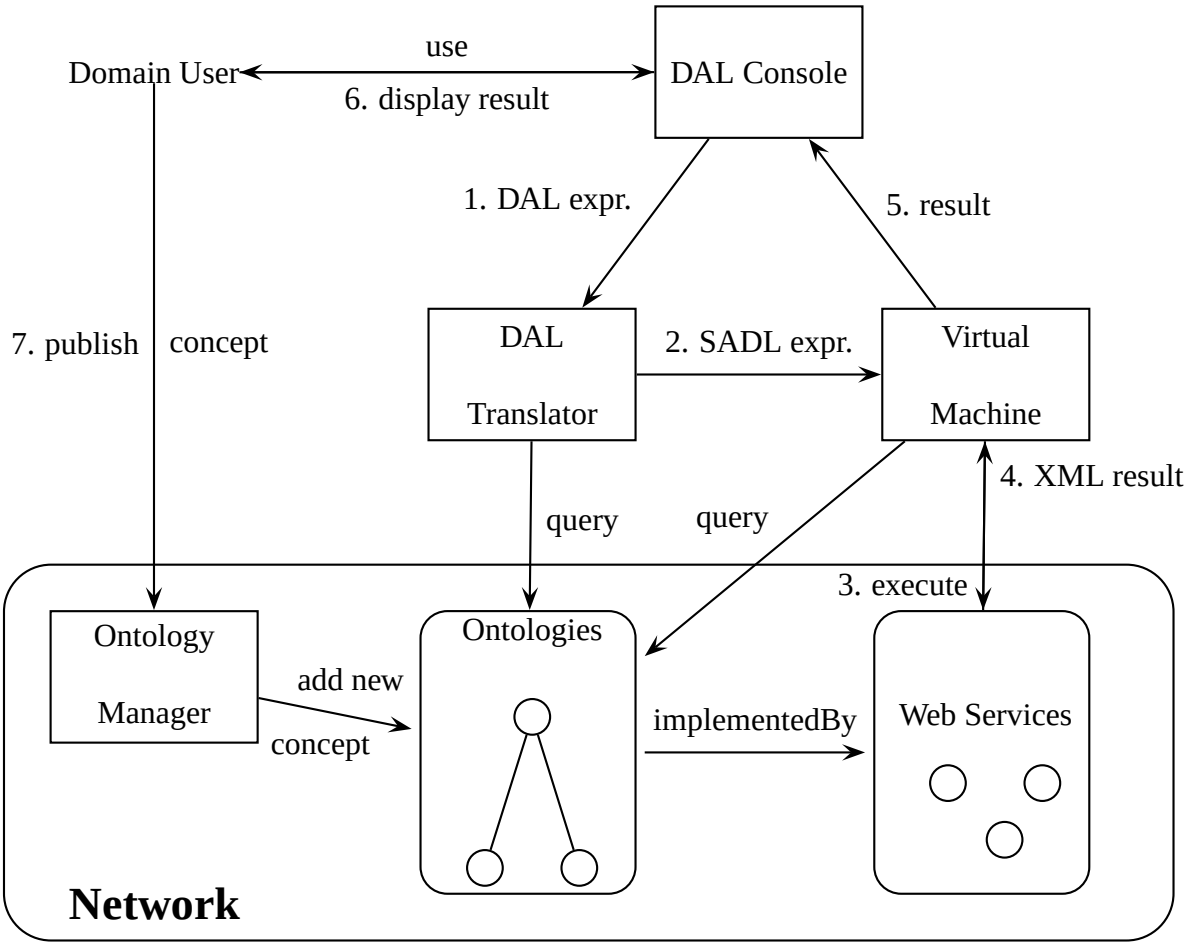

Figure 1.2: DALSystem Architecture

3. A translator to map the domain algorithmic language to the DDVM execution language, SADL (Rus & Curtis 2006), called DAL Translator.
4. A tool for the computer user to interact with her domain ontology and DDVM, called DALConsole.
5. A tool to import solution algorithms into the domain ontology to create new concepts so that the domain ontology can evolve, called OntologyManager.

The system architecture in Figure 1.2 shows how these components are organized and work together. In this system the user interacts with her concepts via the DAL Console. The DAL Console component receives a DAL expression from the user, and send it to the DAL Translator to translate it to the intermediate language called SADL. The DAL Translator queries domain ontologies during the process of translating concepts in the user's DAL expression into SADL instructions. This SADL output will then be sent to the DDVM for execution. The result from DDVM component is displayed back to the user on the DAL Console. This thesis provides a crude implementation of these software tools in a system called DALSystem.

The rest of this thesis is laid out as follows. Chapter 2 reviews and compares important related work with our approach. Chapter 3 shows the process of defining domain ontology and associating domain concepts with their computer implementations. Chapter 4 describes what a domain algorithmic language (DAL) is and how to construct one for the arithmetic domain. The Domain Dedicated Virtual Machine (DDVM) for web execution is described in Chapter 5. In Chapter 6, algorithms for translating a domain expression to an intermediate language for DDVM, called SADL, are discussed. The process of domain ontology evolution (DOE) is described in Chapter 7. Chapter 8 discusses some implementation details about DALSystem. Finally, conclusions and future work are sketched in Chapter 9. A manual for using DALConsole is presented in Appendix A.

CHAPTER 2
RELATED WORK

Chapter 1 described the topic of this thesis: integrating computers into the human problem solving process by making problem formulation step, algorithm development, and algorithm execution steps easier for computer users, especially domain experts. This chapter will review some of the relevant past research on this topic. This will include:

- work on tools for creating languages for a particular domain, especially scientific domains.
- work on computing by composing web services, since we are currently focusing on using web services as execution platform.

Since these are two broad fields of research and there has been much previous research done, I will not attempt to cover every relevant effort. Rather, I will classify the efforts into groups by typical techniques and present one or two representative works in each group.

2.1 Computational Languages for a Domain

In the field of creating computational languages for a domain, work can be divided into two main camps. The first camp arises from the fact that it is not easy for domain users to learn machine languages or even high level programming languages in order to communicate with computers. Thus the first camp tries to make computational languages as close as possible to natural language so that the languages are easy-to-use for domain users. This camp is known as *Natural Language Programming* (NLP). The second camp, called *Domain Specific Languages* (DSL), doesn't try to mimic natural language, but tries

to be efficient and dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. Languages belonging to the DSL camp are typically programming languages or specification languages. DSLs are normally contrasted with general-purposed programming languages. Both of these camps are examined in the following sections in relation to our approach.

2.1.1 Natural Language Programming

The need for natural language programming appeared since the very beginning of the computer era (Sammet 1966, Miller 1981). We want to communicate with computers using human languages. According to Sammet (1966), to make the bridge between natural languages and programming languages, we can go from either side. The first one is to start from full-scaled natural language and try to handle as much as we can. She called it the *top-down approach*. Another way to tackle the problem which she called the *bottom-up approach* is to start from some artificial language and then make it come closer and closer to natural language. Recently, the former is also called *opportunistic recognition* (Liu & Lieberman 2005*b*) and the latter is also known as *Naturalistic Programming* (Lopes, Dourish, Lorenz & Lieberherr 2003, Knöll & Mezini 2006).

2.1.1.1 Top-down Approach

In the top-down approach, systems such as NaturalJava (Price, Rilofff, Zachary & Harvey 2000), Metafor (Liu & Lieberman 2005*a*), Mathematica 8 (Wolfram 2010) allow users to write programs in pure natural language then using Information Extraction (IE) techniques to extract programmatic meaning out of the user's natural language input. Such

programmatic meaning constructs are then translated into a program of a high leveled programming language such as Java (NaturalJava) or Python (Metafor).

```
1 Create a public method called deq that returns a Comparable.
2 Declare an int called i and initialize it to 1.
3 Declare a Comparable called minValue and initialize it to
4 elements' firstElement cast to a Comparable.
5 Please return minValue.
```

Listing 2.1: A natural language program in NaturalJava.

Such systems are usually not only complicated in the natural processing component such as scanner and parser, but also contain complex heuristic mechanisms to reason on semantic structure to yield a corresponding code model. This approach is lossy in the sense that there may be parts of input information dropped out of the interpretation process if the code generator finds that they are irrelevant. Compared to the bottom-up approach, the top-down approach provides more freedom to the user (Sammet 1966). Thus, these systems created the initial impression to the user that they are powerful enough to handle arbitrary natural language input from the user. However, when the user discovered that the systems are not powerful enough to express complex computation structures, the user felt confused about the boundary of the system capabilities (Myers, Pane & Ko 2004).

Our approach in this thesis is not pure naturalistic programming. We try to avoid such confusion from domain users by limiting the user input to a controlled grammar and a vocabulary provided by the domain ontology. In this sense, our approach is closer to the bottom-up approach presented in the following section.

2.1.1.2 Bottom-up Approach

In the bottom-up approach, systems use an artificial language deriving from programming languages with some supplemented features of natural languages. In other words, their languages are basically programming languages but are coated with syntactic sugar to look like natural languages. Such systems can range from a natural language supplemented programming language such as COBOL, AppleScript, etc., to a more mature domain specific language such as one used by Natural Language Computer (NLC) (Biermann & Ballard 1980) for array and matrix computation.

Natural Language Computer (NLC) (Biermann & Ballard 1980) is a computer programming system developed at Duke University in the 1980s. It can also be considered as a domain-specific language for array and matrix computation. NLC was one of the best systems of its time. Within the domain of matrix computation, the system can understand highly complicated commands such as:

```
1 ‘‘double the largest entry
2       in the first row
3         of the matrix
4           containing the column
5             that was doubled by the second to last command.’’
```

One of the reasons why NLC can handle such a complex command like the above example is because the system employed the augmented transition network grammar (Woods 1970). However, controlling the ambiguity of natural language is always a tough topic in any system. An interesting approach used by NLC to reduce the complexity of the user input is to limit the types of input sentence to imperative sentences only. Biermann & Ballard (1980) put it this way “Most of the sentences processed by the system can be thought of

as imperative verbs with their associated operands." Another restriction NLC put on the user input is that the user may refer only to the data structure seen on the terminal screen and use only simple operations upon them. According to Biermann & Ballard (1980) these tricks help a lot. NLC, however, was made to be deeply integrated with English only.

Pegasus (Knöll & Mezini 2006) was a recent effort in NLP, developed at the Darmstadt University of Technology. According to (Knöll & Mezini 2006), Pegasus can read natural language (source text) and create executable program files from the source text. Similar to NLC system, Pegasus is also a domain-specific language currently focusing on matrix calculation; not a general purpose language. Pegasus offered a remedy to NLC's short-coming of multilingual translation by introducing a new important abstraction layer, called *ideas*. This is a semantic network of ideas with one or more ideas can serve as the context for another idea. So instead of translating directly the AST to computer instructions as in the NLC system, a natural language program in Pegasus is parsed using a context free grammar (CFG), then mapped into this semantic representation as a set of ideas. This set of ideas could then be mapped into different output programming languages such as Java or to another Pegasus program in another natural language. In other words, the ideas network serves as an interlingua between Pegasus' natural languages and programming languages.

Our approach in the DALSystem is very much like Pegasus in this perspective. We believe that an intermediate semantic representation is crucial with the benefits of being multilingual. However, instead of developing a custom format for the semantic representation layer as in Pegasus, we use Description Logic (DL) (Baader, Horrocks & Sattler 2007) with standardized OWL (McGuinness & van Harmelen 2004) file format for our semantic

representation. Using DL as our semantic representation helps us leverage the power of automatic reasoning engines like Pellet[1] and Jena[2]. Moreover, Pegasus requires the user algorithm to be exported to Java to be executable, which we believe makes the execution process more complicated. Finally, there is no notion of evolution in both Pegasus and NLC to allow domain users to build new concepts (ideas) from existing ones and then reuse them in new algorithms.

But, the major difference between DAL and NLP is that DAL is an algorithmic language specific to the domain and it is used by the domain experts. Consequently a DAL is a domain specific algorithmic language which is simple to use by domain experts (because it is their natural language) and it is easily disambiguated using domain knowledge. In other words, DAL mimics the domain reasoning not the natural language reasoning. If the domain is the "natural language" then DAL of natural language would probably be mimicking the natural language reasoning. But it still will not be the natural language. The price to be payed is the language generality. Since DAL is dedicated to a domain it is not a general purpose programming language. Only domain experts are supposed to use the DAL of the domain. This reflects the division in the scientific world: domain experts of a domain D_1 (say chemistry) use the language of D_1 in their problem solving process, while domain experts of another domain D_2 (say mathematics) use the language of D_2 in their problem solving process. D_1 and D_2 may share concepts but language expressions of D_1 problem solving algorithms are different from the language expressions of D_2 problem

[1] http://clarkparsia.com/pellet/

[2] http://jena.apache.org/

solving algorithms.

2.1.2 Domain Specific Languages

Even though having the the same goal as NLP systems of being more friendly to domain users, Domain Specific Languages, unlike languages used in NLP systems, don't try to mimic natural languages. They focus on efficient representation and expressive power for a particular domain. Deursen, Klint & Visser (2000) defined a domain-specific language (DSL) as "a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain". As you can see, the use of DSLs for problem solving is not new. According to (Mernik, Heering & Sloane 2005), some of the first DSLs can be found as early as 1957 and 1959, such as APT (Ross 1978) (developed in 1957), a DSL for programming numerically-controlled machine tools, or BNF (Backus 1959), a famous DSL for formal language specification. Since 2000, hundreds of DSLs have been in existence (Deursen et al. 2000). Some of the well-known examples are LEX, YACC, Make, SQL, BNF, and HTML. The areas of their domains are extensive. Among them, Deursen et al. (2000) reported the following groups:

- Software Engineering: Financial products, behavior control, software architectures, databases.
- System Software: Description and analysis of abstract syntax trees, data structures, video device driver specification.
- Multimedia: Web Computing, image manipulation, 3D animation.

- Telecommunications: String and tree languages for model checking, communication protocols, etc.
- Others: Simulation, robot control, solving partial differential equations, digital hardware design.

Mernik et al. (2005) showed the design patterns for DSLs, providing guidelines on when and how to develop DSLs. The need for DSL targeting a specific application domain for specific platforms has resurfaced over the years. On the scientific domains, new domain specific languages are still being developed such as Liszt (DeVito, Joubert, Palacios, Oakley, Medina, Barrientos, Elsen, Ham, Aiken, Duraisamy, Darve, Alonso & Hanrahan 2011) for solving partial differential equations, or BIOLOGO (Cickovski 2004) for cellular and tissue level morphogenenis modeling.

On the surface, according to Deursen et al. (2000)'s definition of DSL, each DAL for a particular domain might look like a DSL. However, while both DSL and DAL are intention revealing, there are strong differences between DAL and DSL. DSLs are designed with the goal of focusing on a more efficient representation of the problem domain compared to general purposed programming language. Sometimes they make DSLs look difficult to understand for domain users. In other words, DSLs help computer experts handle problem domain concepts rather than helping domain users handle computer technology. On the other hand, the main goal of our approach is to help domain experts handle computer technology by bridging the semantic gap between the domain concept and its implementation.

This difference in the design goal leads to the following key difference in the semantics of DSL and DAL. That is, the vocabulary and semantics of DSLs are usually fixed or rarely updated due to committee standardization and long duration processes. After the computer experts created the language, there is very little or no direct collaboration between computer experts and domain users on updating the vocabulary of the DSL. In our approach, the vocabulary of the DAL continuously evolves to serve the needs of domain users by adding new concepts to existing vocabulary. Whenever a new primitive domain concept is added, since DAL is a personal language, it only requires the collaboration between the domain user and computer expert to formalize and implement the concept. It is also worth noting that in the current programming paradigms such as object oriented programming, while it is harder to add primitive concepts than in our approach, it is relatively easy to compose derivative concepts from the existing concepts.

Another subtle difference between DALs and DSLs is the semantics of each expression. The meaning of each expression in DSL is strictly defined in the language specification. In our approach the meaning of each domain term is defined in the ontology, so the meaning of each expression depends not only on the language specification but also largely on the state of the domain ontology.

Finally, since we tend to seek universal, standalone and composable concepts and store them in the ontology in machine readable format, the concepts identified and created in our approach can be reused across domains. But programs created by other DSL tend to be usable only within that language for that particular domain. In other words, while a DSL is a programming language, a DAL is an algorithmic language, independent of the

computer which will execute the DAL algorithms.

These three characteristics also distinguish our approach to domain language from other general purposed programming languages such as Java, C.

2.2 Web Service Composition

In this thesis, web services were chosen as the execution platform because of their interoperability across networks. Therefore it is worth examining previous work on languages for web services composition. Recently as the number of organizations providing their services in the form of web services increases, composition of web services has received more and more interest to support business-to-business integration. Therefore it is not surprising that the literature on web service composition is extensive (Rao & Su 2004, Srivastava & Koehler 2003).

In the literature, there are currently two independently main approaches for composing web services: composition using workflows and composition using AI planning. In the workflow approach, the composition process is mostly done syntactically and manually using XML standards such as WSDL, SOAP, UDDI, BPEL (Margolis 2007). Whereas in the AI planning approach, web services and their constraints (pre-conditions, post-conditions) are specified in Semantic Web languages such as DAML-S or OWL-S (Martin & et al 2003). Then the user only has to specify the goal in the form of a template, the composition process is done automatically via reasoning techniques by a planning engine. The workflow approach is preferred in the business world, while the AI planning approach receives more interest from the academic community.

More recently a hybrid approach has appeared, which tries to combine the strengths from both worlds (Agarwal, Dasgupta, Karnik, Kumar, Kundu, Mittal & Srivastava 2005). In this hybrid approach, the composition process is divided into two phases: logical composition and physical composition. In the logical composition phase, users specify the composition using workflows. Then during the physical composition phase, a composition engine applies AI planning techniques to find out the best combination of underlying web services with respect to some objectives like cost, speed, etc. Our approach for the DALSystem is closely related to the hybrid approach.

I will review these three approaches in the next sections, especially the hybrid approach in comparison to our approach.

2.2.1 Web Service Composition using Workflows

This approach is mainly employed in the business world, where carefully planning and strict security policies are required. A number of XML-based standards such as WSDL, SOAP, UDDI have been developed over the years to formalize the specification, execution protocol and registry of web services. There are currently several web services workflow specification languages, e.g. IBM's BPEL4WS (Andrews & et al. 2003). Such languages provide programming-language-like constructs (sequence, branch, loop) for IT experts to compose web service workflows manually. However such languages are fairly complex, intended to be used by IT experts (developers) not by domain users like domain scientists. In BPEL4WS programs, domain intention is often buried deeply among irrelevant IT concepts such as port, signal, messages, etc.

The syntax of BPEL4WS is so complex that there are even efforts to make that syntax less difficult for IT experts such as SimBPEL (Boisvert, Arkin & Riou 2008) and BPELScript (Bischof, Kopp, van Lessen & Leymann 2009). BPELScript converts the verbose XML syntax of BPEL4WS to a Javascript style language. Even though BPELScript makes BPEL4WS programs easier to understand for IT experts, such programs are still far from understandable for domain users. This is because BPELScript follows BPEL4WS closely to be fully compatible with it.

2.2.2 Web Service Composition using AI Planning

There are efforts from AI community to make the web service composition task less painful to domain users by automating lower level wiring tasks among web services. McIlraith & Son (2002) presented a method to automatically compose web services by applying logical reasoning techniques on a user-predefined template. In this approach, web service capabilities are annotated in DAML-S/RDF at first, then compiled into a situation calculus representation (Narayanan & McIlraith 2002) in Golog, a logical programming language. The user then inputs her goal as a template into the system. For example, a travel reservation procedure using Golog is shown in Listing 2.2.

```
1  proc(travel(D1, D2, O, D),
2  [
3      [ bookRAirticket(O, D, D1, D2),
4        bookCar(D, D, D1, D2)
5      ] |
6      bookCar(O, O, D1, D2),
7      bookHotel(D, D1, D2),
8      sendEmail,
9      updateExpenseClaim
10 ]).
```

Listing 2.2: Travel reservation procedure using Golog. O, D, D1, D2 are Origin, Destination, Departure time, Return Time respectively.

Given the user template, their Golog reasoner evaluates non-deterministic choices and executes the plan on the network using annotated web services. Similar work can be found in (McDermott 2002). We agree with McIlraith & Son (2002) that ontologies of web services should be used to encourage reuse of vocabulary, and shared semantic understanding. However, there is a difference between our approach and their approach on the philosophy of how to conceptualize the domain ontology. Their approach seems to group existing web services into common concepts while in our approach, the concepts are formalized by domain experts first and then IT experts implement these concepts using web services.

Even though much of the burden on low level IT concepts was taken off of the domain user's shoulders, there are still some problems with this approach. That is, the domain users now have to use the declarative style of a logical programming language (Golog) to express their computation. There is a problem about *closed world assumption* with Golog from a web service composition perspective. That is with truth literals we cannot express that new information has been acquired (Rao & Su 2004). For example, one service requester might want to say that a new identity number will be generated and returned from a call to a web service; then will be used during later communication as an ID. Such requirements are very common in both business processes and scientific algorithms.

2.2.3 Web Service Composition using Hybrid Approach

Agarwal et al. (2005) proposed an integrated system using a hybrid approach to web services composition, where the composition is divided into two steps: logical composition and physical composition. In this system, web services are also described formally using domain-specific terminology in a *domain ontology*. When the user wants to compose new services, she writes a *service specification* and provides that to *Logical Composer* module. The Logical Composer will generate an abstract BPEL workflow based on the information from the domain ontology. The abstract BPEL workflow will then be passed to *Physical Composer* to generate a *concrete* BPEL workflow based on some *quantitative* criteria. While the idea sounds similar to ours, there are some differences between their approach and our approach. First of all, their service specification language is a general purposed language for service composition instead of being domain specific like our approach. It also seems that there is no data manipulation and data composition in their language. Moreover, their language only supports a limited set of control flows like sequences, branches, but no loops, while our language is able to handle all typical control flows including loops. Finally, there is no clear framework for domain users to evolve automatically the domain ontology when new composite concepts are added.

Another line of work in the hybrid approach, which is very popular among the scientific community, is implemented in the Grid systems such as Pegasus (Deelman, Singh, hui Su, Blythe, Gil, Kesselman, Mehta, Vahi, Berriman, Good, Laity, Jacob & Katz 2005), Taverna (Oinn, Addis, Ferris, Marvin, Carver, Pocock & Wipat 2004), Kepler (Altintas, Berkley, Jaeger, Jones, Ludascher & Mock 2004, Krishnan & Bhatia 2007), Triana (Taylor,

Wang, Shields & Majithia 2005), WCT (Gubala, Bubak, Malawski & Rycerz 2006), ASKALON (Qin & Fahringer 2008), etc. In this group of systems, domain ontologies are used extensively not only to formalize domain knowledge, but also to support cross-domain interaction. The use of ontologies also enables the domain users to compose work-flows at the level of data meaning and action functions (concepts). In some cases (Qin & Fahringer 2008), ontologies also allow users to semi-automatically compose data flow and perform automatic data conversion.

The common architecture of these systems consists of:

- a GUI workbench to allow domain scientists to compose workflows in drag-and-drop manner.
- an intermediate abstract representation language, normally written in the form of XML such as Sculf (Taverna), AGWL (ASKALON), etc, for these workflows using the domain concepts in the ontology.
- a workflow mapper will then map the abstract workflow into a concrete executable representation using the information from the ontology for looking up available web services.
- an execution engine, e.g. Freefluo (Taverna), ASKALON runtime system (ASKALON), will receive the concrete executable representation of the workflow and run it on the Grid.

The main difference between these systems and the approach in (Agarwal et al. 2005) lies in the use of the GUI workbench for designing workflows instead of using textual repre-sentation, which is supposed to help domain scientists (Scanlan 1989, Kiper, Auernheimer

& Ames 1997). Nonetheless, there is also research found that visual programming might not always be more suitable than textual programming (Green & Petre 1992, Petre 1995).

Compared to our approach, these systems still offer a one-size-fits-all solution for all domains (Curcin & Ghanem 2008), while our approach emphasizes a domain-specific solution for each domain or group of domains. In other words, our approach brings intention-revealing style to web service composition. In a more subtle comparison about the ontology design, most of these workflow systems only focus on the processes (action concepts) and don't pay much attention to data concepts. In (Qin & Fahringer 2008), data concepts receive more attention when the authors separate data concept and data representation so that the domain users don't have to worry about lower level representation of their concepts. However, their approach doesn't provide mechanisms to compose new data concepts from existing data concepts as in our approach. Moreover, Qin & Fahringer (2008) seems to mix the domain expert's view with the IT expert's view about domain concepts in a single ontology level which leads to an inconsistent ontology. In our approach, these two different views are separated clearly into two ontology levels.

Furthermore, only Taverna (Oinn et al. 2004) provides a clear mechanism and tools for the user to share Sculf workflows among scientists as web services. However Taverna doesn't automatically import the concept associating with the workflow into the user's ontology like in our approach for ontology evolution.

Finally my research is part of a bigger theme of developing software for non-expert computer users proposed by Rus (2008). This research focuses on the development of abstractions that liberate computer users from programming. This means that we advocate

the creation of languages dedicated to the problem solving process in the problem domain, not necessarily to program development using conventional programming languages. Previous experiments reported in (Rus & Curtis 2006, Rus & Curtis 2007, Curtis, Rus & Jensen 2008, Rus & Bui 2010) provided software tools based on distributed process execution under the Unix system. My PhD thesis contributes to this research by

- using a domain ontology to formalize a subset of the arithmetic domain,
- implementing a DAL for the arithmetic domain,
- implementing a stack-based domain dedicated virtual machine (DDVM) (Rus 2013) executing on web services,
- providing a mechanism for domain users to evolve their ontology,

in the DALSystem as a proof of concept.

CHAPTER 3
PROCESS OF DEFINING DOMAIN ONTOLOGY

Using ontologies to formalize domain concepts is by no means new. However, much of the current work on ontologies focuses on development and modeling (Welty & Guarino 2001, Gasevic et al. 2009). This thesis, on the other hand, concentrates on structuring domain ontologies to support the automation of concept execution using web services and concept evolution including action concepts and data concepts in a process called Computational Emancipation of Application Domain (CEAD) (Rus 2008). In addition, there is also research on using ontologies for web services composition in scientific workflows such as (Qin & Fahringer 2008, Altintas et al. 2004). As discussed in chapter 2, their approach seems to mix domain expert view and IT expert view of domain concepts in a single ontology level, while we separate these two often different views into two complementary ontology levels, i.e. *pure domain ontology* for domain expert's view and CEAD-ed domain ontology for IT expert's view.

Therefore in this chapter I will briefly discuss the method that we use to formalize a domain vocabulary in a pure domain ontology. Then I focus on discussing our meta-ontology, called CEADOntology, which facilitates the association of each concept in the domain ontology with computational artifacts implementing it. The purpose of this process is to increase the efficiency of executing DAL algorithms by automating the process of searching for web services in the CEADOntology file named OWL(DAL) and evaluating them.

3.1 Domain Ontology

Ontology is a discipline of philosophy dealing with object existence, structures, properties and their relationships. The philosophical work can be traced back to Aristotle in the form of metaphysics. However, the term *ontology* was believed to be coined by Rudof Gockel in 1963 (Welty & Guarino 2001). Ontology found its way to computer science in the early 1980s when AI researchers realized the importance of work in ontology for knowledge representation (McCarthy 1980). The term became a buzzword in knowledge management and enterprise modeling, where "knowledge sharing" and interchange is emphasized.

In this thesis, ontology is used to conceptualize the vocabulary of a problem domain in the CEAD process. It is the first step of the CEAD process where domain ontology is developed by domain experts by gathering domain terms, their properties and relationships. We found that OntoClean (Welty & Guarino 2001) is an efficient methodology for ontology development. OntoClean helps domain experts build domain ontologies by analyzing taxonomies to form *well-founded* ones, called *backbone taxonomies*. A backbone taxonomy consists of only rigid concepts, which are divided into three kinds: *categories*, *types* and *quasi-types*. This backbone taxonomy is specified by a collection of disjoint trees whose nodes are primitive concepts of the domain and whose edges are relationships interpreted as logical subsumptions, i.e., if concept C_1 subsumes concept C_2 then $\forall x.C_1(x) \rightarrow C_2(x)$. After constructing the backbone taxonomy, domain experts can add other kinds of concepts such as *attributions* and *formal roles* which can be combined with primitive concepts in the backbone taxonomy to form lower level concepts such as *mixins* and *material roles*.

Such additional concepts transform the backbone tree to a directed acyclic graph (DAG). Figure 1.1 shows an example ontology for the arithmetic domain.

We choose to use Description Logics (DL) (Baader et al. 2007) as the formal specification of ontologies via Web Ontology Language (OWL). In DL, problem domain terminologies can be captured using the following important types of entities: *concept, role* and *individual*. For a given domain we have a collection of terms $C = \{c_1, c_2, \ldots\}$ representing basic *concepts* of the domain, a set of relations (called *roles*) $R = \{R_1, R_2, \ldots\}$ representing fundamental properties of domain concepts, and a set of individuals $I = \{i_1, i_2, \ldots\}$ representing instances of concepts in the domain. The set-theoretic model of DL allows us to reason about domain objects. The above example ontology for the arithmetic domain in Figure 1.1 is represented in DL using the OWL language as shown in Listing 3.1.

In our approach, the domain ontology modeling the problem solving process consists of two parts: a part that represents the user own ontology (UOO) and a part that represents the domain expert ontology (DEO). The domain expert ontology is built by domain experts using a small taxonomy chosen from a textbook. This ontology is the result of the collaboration between the domain expert and the computer expert as follows:

1. Domain expert defines terms, declares axioms and subsumes hierarchy.
2. Computer expert constructs OWL files from the terms given by the domain expert. Protege can be used to create and edit OWL files.

The User Own Ontology (UOO) is built by extending the DEO. Initially, the UOO is the same as the DEO. Then, during the problem solving process, the UOO is automatically evolved with new concepts representing problems and solution algorithms developed by

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

  <!-- Numbers -->
  <owl:Class rdf:ID="Number"/>
  <owl:Class rdf:ID="Complex">
    <rdfs:subClassOf rdf:resource="#Number" />
  </owl:Class>
  <owl:Class rdf:ID="Real">
    <rdfs:subClassOf rdf:resource="#Complex" />
  </owl:Class>
  <owl:Class rdf:ID="Rational">
    <rdfs:subClassOf rdf:resource="#Real" />
  </owl:Class>
  <owl:Class rdf:ID="Irrational">
    <rdfs:subClassOf rdf:resource="#Real" />
  </owl:Class>
  <owl:Class rdf:ID="Integer">
    <rdfs:subClassOf rdf:resource="#Rational" />
  </owl:Class>

  <!-- Operators -->
  <owl:Class rdf:ID="Operator"/>
  <owl:Class rdf:ID="Unary">
    <rdfs:subClassOf rdf:resource="#Operator" />
  </owl:Class>
  <owl:Class rdf:ID="Binary">
    <rdfs:subClassOf rdf:resource="#Operator" />
  </owl:Class>

  <Unary rdf:ID="unarySubtract" />
  <Unary rdf:ID="factorial" />
  <Binary rdf:ID="addI" />
  <Binary rdf:ID="subtractI" />
  <Binary rdf:ID="multiplyI" />
  <Binary rdf:ID="divideI" />
</rdf:RDF>
```

Listing 3.1: OWL file for arithmetic ontology in Figure 1.1

a particular computer user. Thus the user's ontology space at any given time consists of the core DEO-s, that is commonly available to all the users, and a private part (UOO), which is specific to a given user. The domain expert ontology can also evolve by adding new concepts and importing useful concepts from private ontologies of domain users. The domain evolving process will be discussed later in Chapter 7.

3.2 CEAD Ontology

In this section, I will discuss our meta-ontology, called CEAD ontology, which facilitates the process of associating a problem domain concept with its corresponding computation artifacts. Figure 3.1 shows the overview of this ontology. CEAD ontology can be seen as the complementary view of computer experts to the domain concepts. In other words, this ontology serves as the bridge between the domain concepts and computational artifacts implementing them. There is an important observation here; that is, not all domain concepts have computational meaning. Moreover, there are also concepts in CEAD ontology that are strictly for supporting implementation, which should not be of concern to domain experts. In our approach, to capture the computational essence of a domain ontology, we have developed two main concepts in this meta-ontology: *DataConcept* and *ActionConcept* which are described as follows.

DataConcept is the class of data concepts in a problem domain for example `ari:Integer`, `ari:Real`, `ari:Complex`, etc. in the arithmetic domain. It provides general description for domain data concepts. *DataConcept* class has two subclasses, *PrimitiveDataConcept* and *DefinedDataConcept*. *PrimitiveDataConcept* class consists of data

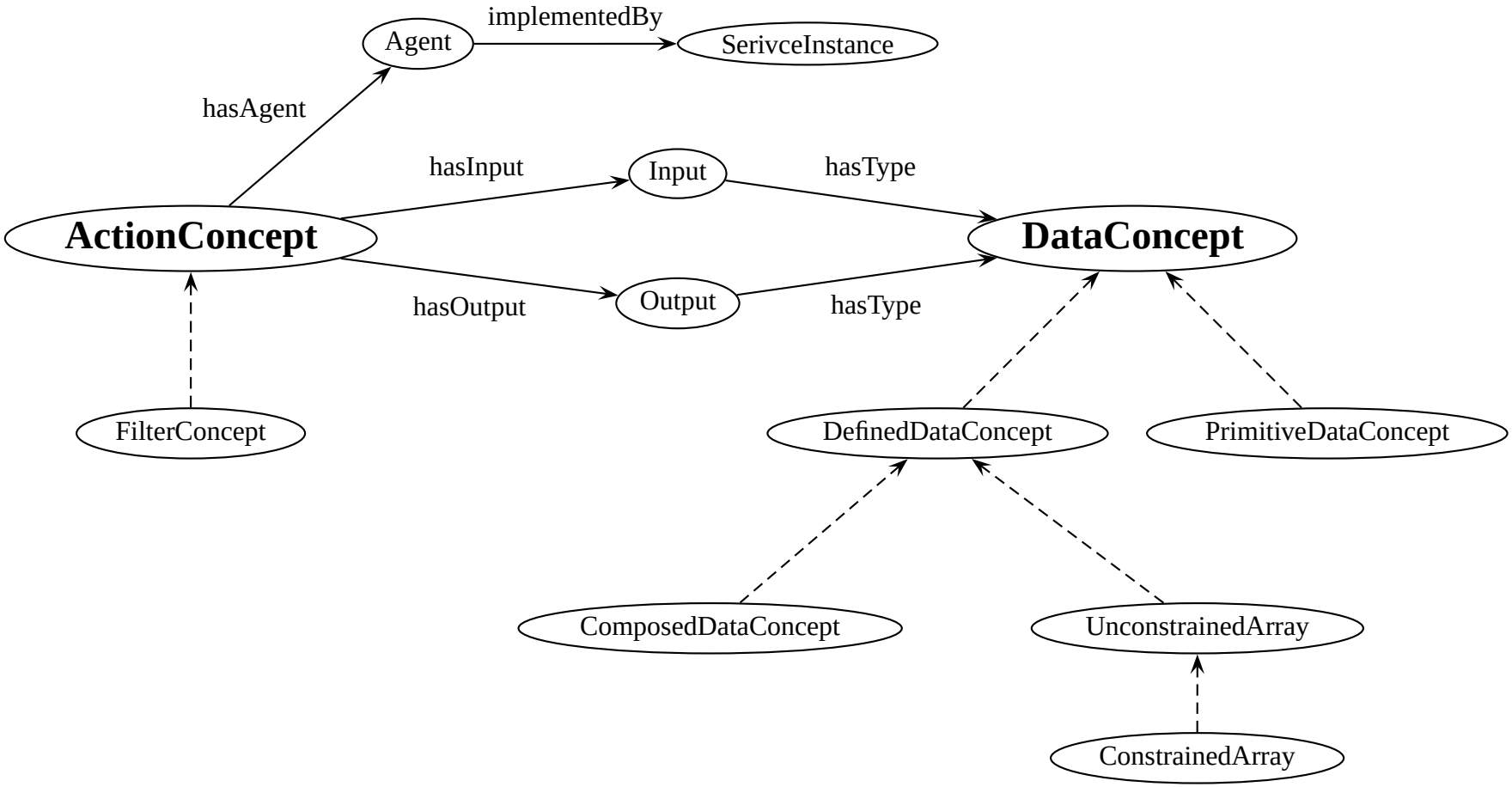

Figure 3.1: Overview of CEAD Ontology

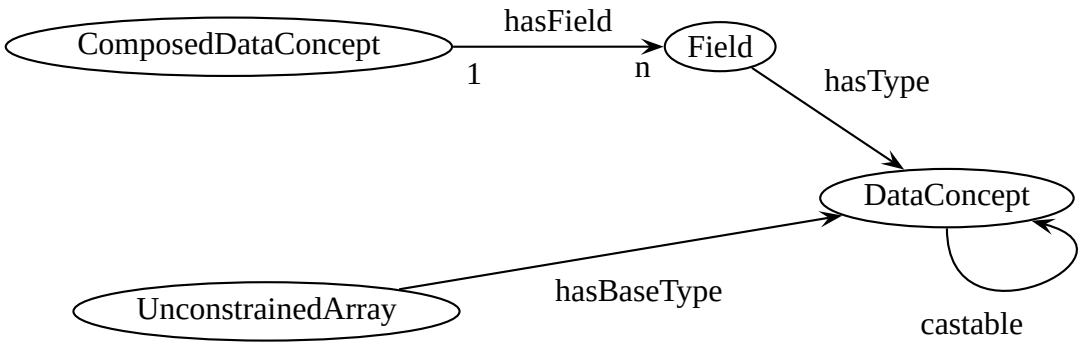

Figure 3.2: Object properties among CEAD concepts for data modeling

elements that have direct XML data types implementing them. For example, `ari:Integer` and `ari:Real` can be represented by `xsd:int` and `xsd:double`, respectively. On the other hand, *DefinedDataConcept* class contains data concepts that have no direct representation and are defined via other data concepts. For instance, `ari:Complex` can be seen as a data concept composed of two `ari:Real` numbers. One is the real component and the other is the imaginary component of that complex number. There are two types of *DefinedDataConcept*, i.e. *ComposedDataConcept* and *UnconstrainedArray*. Each *ComposedDataConcept* has a number of *Field*-s, represented by the property *hasField*. Each *Field* has its type, a *DataConcept*, represented by the property *hasType*. The *UnconstrainedArray* is used to model a data concept which is an array (or a list) of elements of another data concept. Each *UnconstrainedArray* has a type for its elements of *DataConcept*, represented by the property *hasBaseType*. The relationships among these classes of concepts via their

object properties [1] are shown in Figure 3.2.

On the action side, *ActionConcept* is used to model action concepts in a problem domain. An action concept models an agent which performs some computation and transforms its input (data concept) into an output (also a data concept). The relationships among CEAD concepts that facilitate domain action concepts are shown in Figure 3.1. Each *ActionConcept* has a number of *Input*-s, represented by the property *hasInput*. The output of an *ActionConcept* is represented by the property *hasOutput*. Both *Input* and *Output* have their types as *DataConcept*-s, represented by the property *hasType*. Each *ActionConcept* has an agent to manage its computation artifacts which are *ServiceInstance*-s in this case. These relationships are represented by the properties *hasAgent*, *implementedBy*. Finally, *FilterConcept* is a subclass of the *ActionConcept* which converts one data concept to another. The following SPARQL (DuCharme 2011) reasoning rules reflects the relationship between *Filter* and *DataConcept*.

```
1 [rule1: (?a rdf:type cead:FilterConcept) (?a cead:hasInput ?b)
2 (?b cead:inputType ?c) (?a cead:hasOutput ?d)
3 -> (?c cead:castable ?d)]
4
5 [rule2: (?a cead:castable ?b) (?b cead:castable ?c)
6 -> (?a cead:castable ?c)]
```

Listing 3.2: Reasoning rules about *castable* property

These rules say that a type A is castable to a type B if there exists a filter with the input of type A and the output is type B. This castable property is also transitive. This means that if A is castable to B, B is castable to C, then A is castable to C. These reasoning rules are used by DALTranslator in section 6.1.5 and section 6.1.3 to find out if one type of data

[1] http://www.w3.org/TR/owl-features/

Data Properties	*XSD Type*	*Description*
DataConcept		
dataType	xs:anyURI	the URI of XSD type representing this data concept
ConstrainedArray		
hasLowerBound	xs:int	lower bound for a constrained array
hasUpperBound	xs:int	upper bound for a constrained array
Input		
inputName	xs:string	name of the input parameter
order	xs:int	position of the input parameter in the input list
ServinceInstance		
serviceName	xs:string	name of the web service
wsdlFile	xs:string	URI of the WSDL file of the web service
operationName	xs:string	name of the corresponding operation which performs the action concept
portName	xs:string	name of the port on the web service mentioned in the WSDL file

Table 3.1: Data properties for CEAD concepts

concept can be castable to another type of data concept during the process of disambiguating DAL concepts in DAL expressions. Finally, table 3.1 shows data properties of CEAD concepts. The whole OWL file defining CEAD Ontology is shown in Appendix B.

3.3 Associating Domain Concepts with Web Services

Using the CEAD ontology, an IT expert can work with a domain expert to associate domain concepts with their computational artifacts. The IT expert should start with helping

the domain expert identify data concepts and action concepts[2]. Data concepts represent data that can be used in a computational process such as the input and output of such a process. Computational processes are represented as action concepts. Data concepts are associated with XML Schema data types via the property `dataType`. For example, the `Integer` concept is associated with `XSD:int`[3] type, i.e. `Integer` $\xrightarrow{dataType}$ URI(xsd:int). An OWL excerpt for `Integer` definition is shown in Listing 3.3.

```
1 <cead:DataConcept rdf:about="#Integer">
2     <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"
          />
3     <cead:description> Integer concept of arithmetics domain </
          cead:description>
4     <cead:dataType>
5         xsd:int
6     </cead:dataType>
7   </cead:DataConcept>
```

Listing 3.3: Data concept `Integer` definition in OWL

The CEAD process associates action concepts such as `add, subtract, multiply`, etc., with web services which implement them via a Concept Agent. There could be several web services instances that implement the same concept so that if one instance is unavailable, other instances can take over its responsibility. For example, the concept `add` may have the agent `addAgent` implemented by two web service instances: `addInstance1, addInstance2`, where `addInstance1`'s properties are populated in Table 3.1. The agent maintains a list of web services which it can execute as implementations of the action it performs. The RDF triples that define an action concept `X` follows the pattern:

[2] As discussed previously, there might be concepts in the domain ontology which have no computational meaning thus cannot be categorized as either data concept or action concept.

[3] http://www.w3.org/2001/XMLSchema#int

Data Properties	*Value*
ServinceInstance	
serviceName	CalculatorImplService
wsdlFile	http://bula1.cs.uiowa.edu:8282/ CalculatorImplService/CalculatorImpl?wsdl
operationName	addNumbers
portName	CalculatorImplPort

Table 3.2: addInstance1 properties

`X` $\overset{hasAgent}{\longrightarrow}$ `aAgent` and `aAgent` $\overset{implementedBy}{\longrightarrow}$ `aInstance1`; ...; `aAgent` $\overset{implementedBy}{\longrightarrow}$ `aInstanceN`. For example, the `add` concept is represented by the following RDF triples: `add` $\overset{hasAgent}{\longrightarrow}$ `addAgent`,`addAgent` $\overset{implementedBy}{\longrightarrow}$ `addInstance1`. The input and output relation between action concepts and data concepts are represented by the properties `hasInput, hasOutput`. For example: `add` $\overset{hasInput}{\longrightarrow}$ `Integer`, `add` $\overset{hasOutput}{\longrightarrow}$ `Integer`. The representation of these relations are expressed in the OWL language as shown in Listing 3.4.

3.4 Discussion

Pure domain ontology reflects how domain experts see the world. It could be very different from IT experts' view of that world. For example, in the arithmetic domain, a mathematician sees the relationship between `Complex` and `Real` as a subsumption. On the other hand, a computer expert sees both `Complex` and `Real` as instances of the meta-class `DataConcept` where an instance of `Complex` is a record of two `Real` numbers, while `Real` is a primitive concept. So obviously an instance of `Real` is not a record, thus

```
1  <cead:ActionConcept rdf:about="#add">
2    <cead:description>This is the add operation in the
       arithmetics domain.
3      It takes two integers and return the sum of them.
4    </cead:description>
5    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"
       />
6    <cead:hasInput>
7      <cead:Input rdf:ID = "addI1">
8        <cead:inputType rdf:resource="#Integer"/>
9        <cead:order>1</cead:order>
10     </cead:Input>
11   </cead:hasInput>
12   <cead:hasInput>
13     <cead:Input rdf:ID = "addI2">
14       <cead:inputType rdf:resource="#Integer"/>
15       <cead:order>2</cead:order>
16     </cead:Input>
17   </cead:hasInput>
18   <cead:hasOutput rdf:resource="#Integer"/>
19   <cead:hasAgent>
20     <cead:Agent rdf:ID="addAgent">
21       <cead:implementedBy rdf:resource="#addServiceInstance1"
           />
22     </cead:Agent>
23   </cead:hasAgent>
24 </cead:ActionConcept>
25 <cead:ServiceInstance rdf:ID="addServiceInstance1">
26     <cead:wsdlFile rdf:datatype="http://www.w3.org/2001/
         XMLSchema#string">
27       http://bula1.cs.uiowa.edu:8282/CalculatorImplService/
           CalculatorImpl?wsdl
28     </cead:wsdlFile>
29     <cead:serviceName rdf:datatype="http://www.w3.org/2001/
         XMLSchema#string">
30       CalculatorImplService
31     </cead:serviceName>
32     <cead:operationName rdf:datatype="http://www.w3.org/2001/
         XMLSchema#string">
33       addNumbers
34     </cead:operationName>
35     <cead:portName rdf:datatype="http://www.w3.org/2001/
         XMLSchema#string">
36       CalculatorImplPort
37     </cead:portName>
38 </cead:ServiceInstance>
```

Listing 3.4: Action concept add definition in OWL

it cannot be an instance of `Complex`. Thus `Complex` does not subsume `Real`, which contradicts the mathematician's view about the relationship between `Complex` and `Real`.

How is importing ontologies different from importing programming libraries? First of all, the user does not have to install the libraries on her system. The user also does not have to care about where the program is running and the service providers don't have to deploy their code to the user system. These two properties of this model encourage sharing computation resources and knowledge in a competitive environment, which is very important in scientific research communities. Of course, the trade-off for this simplicity in deployment and sharing is the performance. Importing programming libraries into the user's local system offers much better performance since all of the communication happens in the system bus. However, as the speed of the Web becomes faster and faster, this trade-off might not be a problem for most domain users.

Finally, the use of ontology to formalize a domain allows domain users to perform not only syntactic but also semantic searches for the concepts they need. This is very important for the knowledge discovery process since it allows a higher level of expressiveness and gives the user more powerful tools to specify her intent.

CHAPTER 4
DOMAIN ALGORITHMIC LANGUAGE

Given the domain ontology with concepts associated with their computational artifacts, i.e. XML data types or web services, domain users can use these concepts for their computation or compose new concepts using the workflow approach as reported in (Qin & Fahringer 2008, Altintas et al. 2004, Gubala et al. 2006)[1]. Even though such logical composition approaches are better than low-level workflow compositions like BPEL4WS, they are still general purposed tools for workflow composition. Therefore they provide inefficient and unnatural manners to the domain users to express their computation solutions.

In this chapter, I will discuss our approach to the computational language of a problem domain. This language allows domain users to express their computations naturally using terms of their domain without worrying about the representations of these terms in the underlying computer system. We call such language a domain algorithmic language (DAL), where its vocabulary and phrases are characteristic to the domain and domain users can understand it without much explanation. Although this thesis has no ambition of providing a methodology for synthesizing computational languages for all the domains, I still want to present some design guidelines for such languages, based on my experience from working with the domain of arithmetic.

Before further discussion of the design guidelines of these languages, I would like to clarify our assumptions. The main assumption is that solvable problems of any problem

[1]These approaches are equivalent to our lower level language, called SADL, executed by a virtual machine discussed in Chapter 5

domain are expressible in terms of a finite number of well defined concepts. We assume further that these well defined concepts are formalized in a domain ontology as discussed in Chapter 3.

Given the above assumptions, the following guidelines can be used to capture a DAL specification:

1. identify characteristic concepts that are primitive data concepts in the domain ontology and build the lexical rules for such concepts. These concepts are the building blocks of the language. For example, in the arithmetic domain, `Integer`, `Real` are primitive data concepts.
2. identify the rules that combine these basic building blocks to form larger language expressions such as phrases, sentences. These are the second level formation rules, for example, expressions like $1 + 2$, $(a - b) \times c$.
3. identify rules that combine the phrases and sentences to form meaningful solutions to problems, i.e. the discourse level formation rules. Such rules often involve sentences *concatenation*, *choice*, and *iteration*.
4. use a dictionary to provide the link between the domain terms and their meanings in the ontology. The dictionary also helps to reduce the level of ambiguity in the language.

The language should also exploit domain knowledge represented in domain ontology in the forms of subsumption and other kinds of relations for concepts disambiguation.

From the fact that current computer language technology has been providing domain-specific language solutions for hundreds of domains, I believe that most of the domain al-

gorithmic languages can also be handled by conventional language construction tools. As a demonstration, I will show a specification for a DAL of the arithmetic domain in the following sections.

4.1 Examples of DAL(D)

This section shows some examples of DAL for several related domains such as arithmetic, high school algebra, and linear algebra. In each domain, DAL vocabulary, some constructs and discourses are demonstrated.

4.1.1 A DAL for Arithmetic Domain

Vocabulary: The vocabulary for a simple arithmetic domain can consist of: integer, real, add, subtract, multiply, divide, mod, $+, -, *, /, \%$.

Constructs: Some basic constructs are arithmetic expressions as follows: $1 + 2$, $3 * 4$, etc.

Discourse: An interesting discourse (multiple statements) in the arithmetic domain is the Euclidean algorithm to find the greatest common divisor (gcd) of two integers.

```
1  concept: "gcd";
2  description: "This function finds the greatest common divisor (
       gcd) of two integers using Euclidean algorithm.";
3  input: a: integer, b: integer;
4  output: c: integer;
5  local: t: integer;
6
7  while b != 0 do
8    t = b;
9    b = a % b;
10   a = t;
11 endwhile;
12 c = a;
```

4.1.2 High School Algebra

Vocabulary: The domain of high school algebra extends the arithmetic domain with new vocabularies such as: pow (power), sqrt (square root), Complex number, equation, etc.

Constructs: besides constructs from arithmetic domain, high school algebra provides some other constructs such as: sqrt(x), pow(x, y), $3 + i4$, etc.

Discourses: The following algorithm for solving quadratic equation with Complex solution is a typical example for high school algebra.

```
1  concept: "SolverC";
2  description: "This is a quadratic equation solver with complex
       solution.";
3
4  input: a : real, b : real, c : real;
5  output: result: ComplexPair;
6  local: t : real, u : real, x1 : Complex, x2 : Complex;
7
8  t = b * b - 4.0 * a * c;
9  print(t);
10 if t > 0.0 then
11    x1 = constructC((-b + sqrt(t)) / (2.0 * a), 0.0);
12    x2 = constructC((-b - sqrt(t)) / (2.0 * a), 0.0);
13 else
14    u = sqrt(-t) / (2.0 * a);
15    x1 = constructC(-b / (2.0 * a), u);
16    x2 = constructC(-b / (2.0 * a), -u);
17 endif;
18 result.first = x1;
19 result.second = x2;
```

In the above listing, `constructC()` is a concept-constructor which creates a complex number from two real numbers. The first input parameter of `constructC()` is the real part of the complex number it constructs, the second input parameter of `constructC()` is the imaginary part of the complex number it constructs. `ComplexPair` is the concept-

constructor of a pair of two complex numbers.

4.1.3 Linear Algebra

Vocabulary: The linear algebra domain extends the arithmetic domain in a different direction than the high school algebra domain. In this domain, new vocabularies are vector, matrix, linear equation system, and solution.

Constructs: $v[1] = 10$, $v[1] - v[2]$, $v1 * v2$, $m[1][2]/m[1][1]$, where $v, v1, v2$ are vectors, m is a matrix. $v[i]$ means the i-th element of a vector, $m[i][j]$ means the element of i-th row at j-th column.

Discourse: A typical algorithm in the domain of linear algebra is to compute scalar product of two vectors. It is shown in the following listing:

```
1  concept: "product";
2  description: "Compute scalar product of two vectors.";
3  input: a: Vector, b: Vector, n: integer;
4  output: c: real;
5  local: k: integer;
6
7  c = 0.0;
8  for k = 1; if k <= n
9    begin
10     c = c + a[k] * b[k];
11   end
12   withNext k = k + 1;
```

4.1.4 User Dictionary

A user dictionary is where domain terms and their semantics are glued together. Its purpose is very similar to a normal dictionary except for the fact that the meaning of each term is defined by the URI of the corresponding ontological concept in the domain

ontology. The user dictionary is stored in a file which contains the list of entries, one entry for each primitive concept. Each entry has three components separated by a comma ",". They are:

1. wordForm: the term that the domain expert used to denote the concept.
2. category: it could be noun (N) for a data concept, verb (V) for an action concept, adjective (A) for an action concept with the returned value of type boolean.
3. URI: the URI of the concept that the user refers to.

A sample dictionary for arithmetic domain is shown in Listing 4.1. The dictionary is used by the DAL Translator during the semantic annotion process after a DAL expression was parsed. During the semantic annotation process each domain term is mapped to its corresponding domain concept based on the dictionary entries. A more detailed discussion about this annotation process is provided in chapter 6.

4.2 DAL Specification for Arithmetic Domain

Formally a DAL can be specified using a pattern similar to the pattern used to specify computer languages, which consists of a finite set of BNF rules, specifying terms denoting domain characteristic concepts, and few simple BNF rules for statement formation. Further, the DAL specification mechanism should allow both its vocabulary and formation rules to grow dynamically with the domain learning process. We call this the process of DAL's evolution. This allows the domain experts to freely reuse new concepts and solution algorithms as components of the new concepts and solution algorithms developed during the problem solving process.

```
integer ,N, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#Integer
real ,N, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#Real
sqrt ,V, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#sqrt
+,V, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#add
+,V, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#addR
*,V, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#multiply
*,V, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#multiplyR
-,V, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#subtract
-,V, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#subtractR
-,V, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#unarySubtract
-,V, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#unarySubtractR
/,V, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#divide
/,V, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#divideR
%,V, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#modI
>,A, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#greaterThan
>,A, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#greaterThanR
<,A, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#lessThan
<,A, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#lessThanR
<=,A, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#lessThanOrEqual
==,A, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#equalI
!=,A, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#notEqualI
not ,A, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#notOp
and ,A, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#andOp
or ,A, http://bula1.cs.uiowa.edu/owl/arithmetic.owl#orOp
stringarray ,N, http://bula1.cs.uiowa.edu/owl/cead.owl#StringArray
string ,N, http://bula1.cs.uiowa.edu/owl/cead.owl#String
```

Listing 4.1: Dictionary entries for Arithmetic Domain

4.2.1 Rule Representation

Grammar rules for DAL are written in Backus-Naur Form (BNF) form (Backus 1959), except for some lexical rules which are written in the JFlex [2] style for specifying regular expressions.

4.2.1.1 Characters

DAL use ASCII charset with the following special character classes:

```
1 NONNEWLINE_WHITE_SPACE_CHAR = [\ \t\b\012]
2 NEWLINE = \r|\n|\r\n
3 WHITE_SPACE_CHAR = [\n\r\ \t\b\012]
```

4.2.2 Lexical elements

Comments: every line starts with "#" is considered a comment and ignored during the parsing process.

Semicolons: Every statement ends with a semicolon ";".

Identifiers: Identifiers name algorithm entities such as variables and concepts (types). An identifier is a sequence of one or more letters and digits. The first character must be a letter.

```
1 IDENTIFIER = [a–zA–Z]([a–zA–Z0–9]*)
```

Keywords: The following keywords are reserved and should not be used as identifiers.

[2] Available at http://jflex.de/

array	endconcept	import	message	return
begin	endif	input	not	then
description	endrecord	is	of	vocabulary
do	endwhile	local	ontology	while
else	for	localURI	output	withNext
end	if	loop	record	

Operators and Delimiters: The following character sequences represent operators, delimiters, etc.

+	=	!=	(	)
-	<	<=	[	]
*	>	>=	{	}
/	==	..	,	;
%	!		.	:

Integer Literals: An integer literal is a sequence of digits representing an integer constant.

```
1 DEC_INT_LITERAL = 0 | [1-9][0-9]*
```

Floating-point Literals: A floating-point literal is a decimal representation of a floating-point real constant.

```
1 FLOAT_LITERAL  = ({FLit1}|{FLit2}) {Exponent} ?
2
3 FLit1     = [0-9]+ \. [0-9]*
4 FLit2     = \. [0-9]+
5 Exponent = [eE] [+-]? [0-9]+
```

String Literals: A string literal represents a string constant.

```
1 STRING_TEXT = (\\\"|[^\n\r\"]|\\{WHITE_SPACE_CHAR}+\\)*
```

4.2.3 Declarations

Variable Declarations: A variable declaration creates a variable, binds an identifier to it and gives it a type.

```
1 var_dec_cmd ::= IDENTIFIER:name ":" IDENTIFIER:type
```

For example,

```
1 x: integer;
2 y: Vector;
```

Array Declarations: An array declaration creates a variable, binds an identifier to it and gives it an anonymous array type.

```
1 dec_arr_cmd ::= IDENTIFIER:name ":" "array" "(" INT_VALUE:v1 ".."
      INT_VALUE:v2 ")" "of" IDENTIFIER:type
2 dec_arr_cmd ::= IDENTIFIER:name ":" "array" "of" IDENTIFIER:type
```

For example,

```
1 x: array (1 .. 3) of real;
2 y: array of integer;
```

Concept Declarations: A concept declaration binds an identifier, the concept name, to a new concept type of either a record type or array type.

```
1 field_dec_cmd ::= IDENTIFIER:name ":" IDENTIFIER:type ";"
2 field_list ::= field_dec_cmd: cmd | field_list:l field_dec_cmd:
     cmd
3 concept_dec_cmd ::= "concept" IDENTIFIER:name "is" "record"
     field_list:l "endrecord" ";" "endconcept"
4 concept_dec_cmd ::= "concept" IDENTIFIER:name "is" "array" "("
     INT_VALUE:v1 ".." INT_VALUE:v2 ")" "of" IDENTIFIER:type ";" "
     endconcept"
5 concept_dec_cmd ::= "concept" IDENTIFIER:name "is" "array" "of"
     IDENTIFIER:type ";" "endconcept"
```

For example,

```
concept Complex is
    record
        RealPart: real;
        ImgPart: real;
    endrecord;
endconcept

concept Vector3D is
    array (1 .. 3) of real;
endconcept

concept Vector is
    array of real;
endconcept
```

Declaration List: A list of declarations on input, output or local scope.

```
dec_cmd ::= var_dec_cmd:vc | dec_arr_cmd: ac | concept_dec_cmd:
   cc
dec_list ::= dec_cmd:cmd | dec_list:l "," dec_cmd:cmd
```

An example:

```
x: integer, y: real, z: array of real,
concept ComplexPair is
    record
        first: Complex;
        second: Complex;
    endrecord;
endconcept
```

Input Declarations: A list of declarations on the input scope of the algorithm.

Variable declarations in this scope will take the value from the caller.

```
input_decl_opt ::= "input" ":" dec_list:l ";"
```

For example,

```
input: x: integer, y: real, z: Vector3D;
```

Output Declarations: A list of declarations on the output scope of the algorithm. Only one variable declaration should be in this scope. Its value will be returned back to the caller.

```
output_decl_opt ::= "output" ":" dec_list:l ";"
```

For example,

```
output: x: integer;
```

Local Declarations: A list of declarations on the local scope of the algorithm. Variable declarations in this scope will be initialized with default values specified by their types.

```
local_decl_opt ::= "local" ":" dec_list:l ";"
```

For example,

```
local: x: Vector, y: Vector, n: integer;
```

Import Declarations: An import declaration states that the algorithm contains some concepts or vocabularies from the imported ontologies or dictionary. There are two kinds of imports: ontology or vocabulary, as follows:

```
1 import_ontology_cmd ::= "import" "ontology" STR_VALUE:uri ";"
2 import_ontology_cmd ::= "import" "ontology" STR_VALUE:uri "
    localURI" STR_VALUE:luri ";"
3 import_vocab_cmd ::= "import" "vocabulary" STR_VALUE: uri ";"
4 import_vocab_cmd ::= "import" "vocabulary" STR_VALUE: uri "
    localURI" STR_VALUE:luri ";"
5 import_cmd ::= import_vocab_cmd: cmd | import_ontology_cmd: cmd
6 import_list ::= import_cmd: cmd | import_list: l import_cmd: c
```

In these declarations, if a `localURI` is specified, the resource at `localURI` will be loaded instead of the resource at `uri`.

For example,

```
1 import ontology "http://bula1.cs.uiowa.edu/owl/arithemticCEAD.owl
    " ;
2 import ontology "http://bula1.cs.uiowa.edu/owl/arithemticCEAD.owl
    "
3                     localURI "file:../../owl/arithmeticCEAD.owl";
4 import vocabulary "http://bula1.cs.uiowa.edu/owl/Arithmetics.dic"
     ;
5 import vocabulary "http://bula1.cs.uiowa.edu/owl/Arithmetics.dic"
6                     localURI "file:../../owl/Arithmetics.dic";
```

Description Declaration: This optional declaration describes what an algorithm is doing or what type of concept is being created.

```
1 desc_opt ::= "description" ":" STR_VALUE:d ";"
```

For example,

```
1 description: "This is a quadratic equation solver."
```

Algorithm Name Declaration: This is a required declaration for the name of the algorithm or concept. This name will be used by OntologyManager as the concept name in

case this algorithm is added to the ontology. This declaration is always at the beginning of the algorithm. For example,

```
concept: "gcd";
```

4.2.4 Terms (expressions)

Operands: Operands denote the elementary values in an expression. An operand may be a literal, variable, or a phrase.

```
literal ::= STR_VALUE:v | INT_VALUE:v | FLOAT_VALUE:v
term ::= literal:v | IDENTIFIER:id | phrase:p
```

Selectors: For a primary expression x, the selector expression

```
x.f
```

denotes the field f of the value x. For example, given the declarations:

```
local: concept ComplexPair is
    record
        first: Complex;
        second: Complex;
    endrecord;
endconcept,
x: ComplexPair;
```

the user may write:

```
x.first
x.second
```

Grammar rule:

```
term ::= term:t "." IDENTIFIER:id
```

Indexes: A primary expression of the form

```
a[i]
```

denotes the element i-th of the array. The value i is called the index.

Grammar definitions for indexes:

```
term ::= term:t "[" term:i "]"
```

Phrases: Given an term f of action concept F,

```
f(a1, a2, ... an)
```

calls f with arguments $a1, a2, \ldots an$.

As defined in

```
term_list ::= term:p | term_list:l COMMA term:p
phrase ::= IDENTIFIER:id "(" term_list:vl ")"
phrase ::= IDENTIFIER:id "(" ")"
```

Operators: Operators combine operands into expressions.

```
term ::= term:l bin_op term:r | "-" term:l | "not" term:l
bin_op ::= "+" | "-" | "*" | "/" | "%"
           | "==" | "!=" | "<" | ">"
           | "<=" | ">="
           | "and" | "or"
```

For examples,

```
1 x + 1
2 -3
3 a > b
4 c < d
5 (x > y) and (t <= v)
```

Note that unlike programming languages, the semantics of each operator is not defined in DAL but in the ontology via the CEAD process.

4.2.5 Commands (statements)

Assignments:

```
1 assign_cmd ::= term:lhs "=" term:t ";"
```

After an assignment statement, the evaluated value of the term t on the left hand side (LHS) will be stored in the location of the lhs term.

If Command: The grammatical rules for If command is defined as:

```
1 if_cmd ::= "if" term:p "then"  cmd_list:cl "endif" ";"
2 if_cmd ::= "if" term:p "then"  cmd_list:cl "else" cmd_list:cl2 "
    endif" ";"
```

Or we can defined it as:

```
1 if expr then
2    statement1(s);
3 else
4   statement2(s);
5 endif;
```

Where `expr` is a boolean expression. If `expr` is evaluated to true, then `statement1(s)` are executed. Otherwise, `statement2(s)` are executed. The **else** branch is optional.

For example,

```
1 if n < 3 then
2   z = a;
3 endif;
```

While Command:

```
1 while_cmd ::= "while" term:p "do" cmd_list:cl "endwhile" ";"
```

In the **while-loop** construct of

```
1 while expr do
2   statement(s);
3 endwhile;
```

if the boolean expression `expr` is evaluated to `true`, the `statement(s)` is executed and the expression is re-evaluated. This cycle repeats until `expr` becomes `false`.

For example, the Euclidean algorithm for finding gcd of two integers a, b is expressed by the **while-loop** as follows:

```
1 while b != 0 do
2   t = b;
3   b = a % b;
4   a = t;
5 endwhile;
```

For-loop Command:

```
1 forloop_cmd ::= "for" assign_cmd:ic "if" term:lc "begin" cmd_list
    :cl "end" "withNext" assign_cmd:ac
```

The **for-loop** formal syntax can be written as:

```
for expr1; if expr2
  begin
    statement(s);
  end
  withNext expr3;
```

which is equivalent to

```
expr1;
while expr2 do
    statement(s);
    expr3;
endwhile;
```

For example, the addition of two vectors can be written as

```
for i = 1; if i <= 3
  begin
    v[i] = v1[i] + v2[i];
  end
  withNext i = i + 1;
```

4.3 DAL Use

The DAL use is illustrated by two examples. The first example is the DAL expression of the Euclidean algorithm for finding the greatest common divisor of two integers, shown in Listing 4.2. The second example is the DAL expression of the Householder algorithm for finding the solution of a system of linear equations, which is shown in Appendix C. For more detail on how to use this language with the actual DALSystem please refer to Appendix A.

```
1  concept: "gcd";
2  description: "This is function for find greatest common divisor (
       gcd) of two integers using Euclidean algorithm.";
3  input: a: integer, b: integer;
4  output: c: integer;
5  local: t: integer;
6
7  while b != 0 do
8    t = b;
9    b = a % b;
10   a = t;
11 endwhile;
12 c = a;
```

Listing 4.2: Euclidean algorithm for finding GCD of two integers

CHAPTER 5
DOMAIN DEDICATED VIRTUAL MACHINE AND SADL LANGUAGE

After domain concepts were associated with their computational meaning and the computational language for the domain was developed, the automation of user algorithm execution is based on two main software components: (1) A translator that maps user algorithm $\mathcal{A}$ into an intermediate language expression $IL(\mathcal{A})$, called SADL, whose instructions are domain concepts associated with the URL of computational artifacts implementing them, and (2) An interpreter operating on the intermediate language expression $IL(\mathcal{A})$ generated by the translator, executing computational artifacts (web services) encountered at each instruction. The translator can be implemented by conventional compiler construction tools as discussed in Chapter 6. The interpreter can be implemented as a virtual machine, which I will discuss in this chapter in section 5.1. The intermediate language of this virtual machine will be discussed in sections 5.2 and 5.3.

5.1 Domain Dedicated Virtual Machine

The term Domain Dedicated Virtual Machine (DDVM) was coined by Rus (2008) to describe a virtual machine which performs domain user algorithms based on domain concepts implemented by web services. This virtual machine automates the execution of DAL algorithms, which can instead be performed manually by a problem solver using computers as brain assistants, in order to increase the efficiency.

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

- ConceptC is a Concept Counter, that, for a given DAL algorithm $\mathcal{A}$, points to the URI of the concept in the OWL(DAL) to be performed next during the algorithm execution;
- Execute() is the process that executes the computation meaning of the domain concept assigned to ConceptC;
- Next() is a function which determines the next concept of the DAL algorithm $\mathcal{A}$ to be performed by Execute() during algorithm execution.

The DDVM performs similarly with the Program Execution Loop (PEL) ((Rus 1993), p. 129) and therefore the algorithm execution by DDVM can be described by the following *Domain Algorithm Execution Loop* (DAEL) (Rus 2013):

```
1 ConceptC = getFirstDALConcept(DAL algorithm)
2 while (ConceptC is not End)
3     Execute (ConceptC);
4     ConceptC = Next(ConceptC, DALalgorithm)
5 Extract the result and display the final output to the user
```

At the high level view, DDVM is very similar to a Virtual Monitor (Popek & Goldberg 1974). The ConceptC is the counter part of the program counter, and the domain action concept that the ConceptC refers to is similar to the function executed by the OS simulating instructions of the machine implemented by the VM. Finally, Next() is similar to the process that determines the next instruction of the program run by the VM. However, the difference between a DDVM and a Virtual Monitor is that the memory of the machine implemented by DDVM is all the ontologies imported by the DAL algorithm OWL(DAL), and the processor of the DDVM is the collection of all processors available on the Web of

services participating in the OWL(DAL). Therefore, the DDVM is a true domain dedicated virtual machine.

There are a few key things to note. Firstly, all the conceptual instructions of DDVM are abstract, performing the user's solution logically. It is the DDVM that interprets these conceptual instructions and executes the computational artifacts associated with these conceptual instructions at the runtime. This means that the user solution in the form of DDVM conceptual instructions remains useful for a longer period of time; even when the underlying computational artifacts change over time. Thus, this architecture encourages the reuse of user solutions.

Secondly, due to the fact that DDVM makes remote procedure calls to web services implementing action concepts, it essentially executes a distributed algorithm. Since our implementation of DDVM is inspired by the stack based virtual machine, Java Virtual Machine (JVM), we can think of DDVM as a JVM that conceptually runs on top of the network across organization boundaries.

Finally, even though DDVM operates at the same abstraction level as other workflow engines in (Qin & Fahringer 2008, Gubala et al. 2006, Altintas et al. 2004), there are some differences between our work and theirs. These differences result from the fact that DDVM is designed to allow the domain user to manipulate data at much more fine grain level such as allowing access to fields of composed data concepts or to array elements, and declaring variables. Since DDVM is designed to facilitate more lively interactions between domain users and their concepts, we pay more attention to the input/output process of data concepts to human readable form. On the other hand, current existing workflow engines

are designed to facilitate a Grid based environment where there is less interaction between user and the computation process with mainly scheduled jobs. These differences lead us to more of a full-pledged virtual machine while existing workflow engines tend to be more of a simple composing engine.

5.2 Structure of SADL File

The Software Architecture Description Language (SADL) (Rus 2013) was developed by us to represent problem domain solutions in the abstract form. Its goal is similar to the goal of the Intermediate Language (IL) used by Microsoft's ASP.NET Framework for providing a common ground for several higher level programming languages. SADL serves as an Intermediate Language for all the DAL languages designed to run on DDVMs.

SADL language design was inspired mainly by JVM intermediate language while its syntax is based on XML representation. We choose XML for SADL representation because the XML tag set provides a rich and powerful language which is easily expandable by adding new attributes to XML elements without breaking file format. Moreover, there are many existing tools to process XML files.

A SADL file is organized into two sections: *declaration* and *instructions*. The `declaration` section contains information about imported ontologies, input variables, and output variables. Hence, it is divided into three subsections: *imports*, *inputs*, *outputs*. Inside the *imports* subsections are *importOntology* tags which specify ontologies that this algorithm may use. The *inputs* and *output* subsections contains *input* and *output* tags respectively for the declarations of input and output variables. The general structure of the

declaration section is shown in Listing 5.1. We will discuss these tags in detail in Sec-

```
<declaration>
  <imports>
    <importOntology uri="URI(Ontology1)" />
    ...
    <importOntology uri="URI(OntologyN)" />
  </imports>
  <inputs>
    <input type="URI(type1)" index="i1" />
    ...
    <input type="URI(typeN)" index="iN" />
  </inputs>
  <outputs>
    <output type="URI(outputType)" index="o" />
  </outputs>
</declaration>
```

Listing 5.1: Declaration section of SADL file

tion 5.3.

After the declaration section is the section for DDVM conceptual instructions. At this lowest level of SADL is a dynamic collection of primitive terms (instructions) used to denote problem domain concepts such as `Complex, addComplex`, etc, or IT specific terms such as `push, store`, etc. All the instructions are sequential by default. The branching and repetition constructs are implemented by using `jumping instructions` and labels to jump from one place to other in this sequence of instructions. Listing 5.2 shows an example of a SADL file for computing the sum of two real numbers.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <sadl xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <declaration>
4      <imports>
5        <importOntology uri="http://bula1.cs.uiowa.edu/owl/cead.owl
             " local="file:../OntologyManager/src/main/webapp/
             ontologies/cead.owl" />
6        <importOntology uri="http://bula1.cs.uiowa.edu/owl/
             arithemticCEAD.owl" local="file:../../owl/arithmeticCEAD
             .owl" />
7      </imports>
8      <inputs>
9        <input type="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
             Real" index="1" />
10       <input type="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
             Real" index="2" />
11     </inputs>
12     <outputs>
13       <output type="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
             Real" index="3" />
14     </outputs>
15   </declaration>
16   <init type="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#Real"
       index="3" />
17   <load index="1" />
18   <load index="2" />
19   <addR xmlns="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#"
       params="2" />
20   <store index="3" />
21 </sadl>
```

Listing 5.2: Two push instructions for complex data types then adding them together using addComplex concept

5.3 DDVM Conceptual Instructions

In this section, I will provide detailed explanations about the semantics of each instruction of our virtual machine, DDVM. For each instruction, I show its syntax as XML element and the corresponding attributes. Then I explain how the virtual machine should behave for that instruction.

Before discussing the DDVM instructions set, I would like to discuss the internal architecture of our virtual machine. The first component of our virtual machine is the concept counter. At any time, this counter points to the current concept to be executed. After the execution of that concept finishes, the counter moves to the next concept. Since our virtual machine is a stack based one, the second most important component is the stack. This stack is initially empty, and during the virtual machine execution, it holds intermediate results of the computation process. Finally, we use virtual registers to hold the contents of the variables used in user algorithm, including input, output and local variables. The virtual registers are stored in a dynamic array. To access the content of a register, we need to provide the index of that register in the instruction such as *load* and *store*. When the virtual machine is initialized, it loads all the contents of input parameters into input registers, the output register and local registers are initialized to the default values of their types. Finally, the virtual machine maintains an ontology model which loads all the imported ontologies and provides the reasoning service for the virtual machine during the execution. Provided these assumptions about the virtual machine architecture, we can move on to examine the instruction set of this virtual machine.

5.3.1 Declaration Intructions

The first type of declaration instructions is the importOntology instruction. DDVM will load the ontology O at `uri` attribute of the XML tag the URI(O). If the `local` attribute is present, then the content of the URI(O) will be loaded locally with the content of the file at URI(localFile.owl).

```
1 <importOntology uri="URI(O)" local="URI(localFile.owl)" />
```

After loading this instruction, all the classes, individuals and their properties of this ontology are loaded into the DDVM ontology model for later queries. A concrete example of importing the arithmeticCEAD.owl ontology at the URI of `http://bula1.cs.uiowa.edu/owl/arithemticCEAD.owl` is shown below.

```
1 <importOntology uri="http://bula1.cs.uiowa.edu/owl/
    arithemticCEAD.owl"
2                 local="file:../../owl/arithmeticCEAD.owl" />
```

The second type of declaration instructions is the input variable declaration instruction.

```
1 <input type="URI(varType)" index="r" />
```

This instruction has two attributes, `type` and `index`. The `type` attribute is the URI of the type of the variable. The `index` attribute is the index of the DDVM register allocated to this variable. For example, the instruction

```
1 <input type="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#Real"
2        index="2" />
```

means that the DDVM register 2 is allocated for an input variable of type `http://bula1.cs.uiowa.edu/owl/arithmetic.owl#Real`.

Similarly, the output instruction describes the `type` and the DDVM register index of the output variable.

```
1 <output type="URI(varType)" index="r" />
```

Both attributes have the same meaning as those of the `input` instruction.

5.3.2 Virtual Register Traffic

DDVM registers typically hold variable values. Register index starting from 1 are set aside for the algorithm's input variables. The next registers are for output variables. Local variables are assigned to registers after the output registers. DDVM registers are *untyped*, so they can hold any kind of value.

The `load` instruction pushes the content of a DDVM register on top of the stack (TOS). The register value is unaffected by the instruction.

```
1 <load index="r" />
```

The `index` attribute specifies the index of the DDVM register needed to be loaded. Note that the value of each register is a data item, so it contains both value and the type of the value.

The `store` instruction pops a value from the DDVM stack and stores it in the content of the specified DDVM register.

```
1 <store index="r" />
```

The `index` attribute specifies the index of the DDVM register receiving the value. The whole data item content including both value and the type of the value are stored in the register.

The `pushStr` instruction pushes a string value of type `cead:String` on TOS.

```
1 <pushStr value="string" />
```

The `value` attribute holds the value of the string.

The `loadConst` instruction converts the constant value on TOS to the type specified by the instruction.

```
1 <loadConst conceptURI="URI(c)" />
```

The `conceptURI` attribute specifies the URI of the concept to be loaded. The DDVM makes a call to the input filter of concept c to convert the value on TOS to the type of concept c. The received value will be pushed back to TOS.

The `init` instruction initializes the default value for a variable.

```
1 <init type="URI(c)" index="r" />
```

The `type` attribute specifies the type of this default value. The `index` attribute specifies the register which holds the value of the initialized variable.

5.3.3 Action Instructions

This is a special class of instructions of each DDVM. It contains all the action concepts of the application domain. The template of these instructions is shown below.

```
1 <conceptURI params="p" />
```

The `params` attribute specifies the number of input parameters that the DDVM needs to pass to the concept. The result of this call to the action concept will be pushed back to TOS. For example, the call to the concept `multiplyR` of the arithmetic domain looks like:

```
1 <multiplyR xmlns="http://bula1.cs.uiowa.edu/owl/arithmetic.owl
    #"
2             params="2" />
```

That means the DDVM will pop 2 items from the stack and send them to the IT artifact implementing the concept multiplyR for execution. The result is a value of type `http://bula1.cs.uiowa.edu/owl/arithmetic.owl#Real` and will be pushed on TOS.

The execution of this instruction using a web services as the implementation artifact is shown in the following algorithm:

1. The DDVM makes a SOAP call to the remote web service implementing the semantics of the corresponding domain concept.
2. The current execution thread T_1 of the DDVM at machine A is blocked and waits for the result from the remote server at a machine B.
3. The remote server at machine B creates a process T_2 executing the corresponding web services.
4. After the process T_2 finished, the remote server returns the result to the machine A.
5. The machine A notifies the thread T_1 of the DDVM.

6. The thread T_1 receives the result from machine B and pushes it to the TOS (Top of the Stack).

After that the concept counter points to the next instruction.

5.3.4 Field Access Instructions

The `getfield` instruction pushes the value of a particular field of a data item on TOS. The form of a `getfield` instruction follows:

```
1  <getfield field="fieldName" />
```

When encountering this instruction, the DDVM will pop the data item on TOS, then DDVM accesses the field specified by the `field` attribute. The result will be pushed back on TOS. For example, if TOS holds a value of a complex number with two fields `RealPart` and `ImgPart`.

```
1  <getfield field="RealPart" />
```

The instruction pops the complex number out of TOS. The DDVM reads the value of the `RealPart` field and pushes the value back to TOS.

The `putfield` instruction pushes a value to a particular field of a data item on TOS. The syntax of a `putfield` instruction follows:

```
1 <putfield type=URI(c) field="fieldName" />
```

This instruction pops two values from TOS. The first value (v) is the value that should be stored at the field (fieldName) specified by the attribute `field` of the `putfield` instruction. The second value is a reference to the data item (x) with the field to be stored. When

the instruction completes, the value field `fieldName` of x will be v and the stack will have two fewer items.

5.3.5 Branching

There are two kinds of instructions that accommodate conditional branches, i.e. `jumpfalse` and `jumptrue`. The `jumpfalse` instruction expects that the TOS is a boolean value of type `cead:Boolean`.

```
1 <jumpfalse label="labelName" />
```

The branch is taken if the boolean value is true. Otherwise, the DDVM concept counter will jump to the position with the label `labelName`.

The `jumptrue` instruction is similar to the `jumpfalse` instruction with the opposite action. That means, the branch is taken if the boolean value is false. Otherwise, the DDVM concept counter will jump to the specified label position.

```
1 <jumptrue label="labelName" />
```

CHAPTER 6
TRANSLATION FROM DAL TO SADL

The mapping of the DAL algorithms into SADL expressions can be done by the domain expert by hand. This is feasible for toy problems. For more sophisticated problems it is beneficial to automate this process. I have developed a translator that maps DAL algorithms into SADL instructions which are then interpreted by a DDVM. This translator can be implemented using conventional compiler construction tools (Aho, Sethi & Ullman 1986, Fischer, Cytron & LeBlanc 2010) with some modification to take advantage of domain knowledge in the ontology for concept disambiguation and to tailor to DDVM instructions as the target code.

Our translator follows the traditional processing pipeline in compiler design as shown in Figure 6.1. In this pipeline, the Lexical Analyzer is specified by a set of regular expressions, generating a token stream from the DAL expression. The token stream is then passed to the Parser whose grammar is specified by an LR(1) grammar (Knuth 1965) in the form of BNF rules. The Parser generates an Abstract Syntax Tree (AST) from the token stream. The AST is then annotated by the Semantic Analyzer with deeper semantic information on each node such as type information. Finally, the Code Generator receives the Annotated AST and generates the corresponding SADL code.

Among the components of the processing pipeline, we use standard tools for constructing Lexical Analyzer (JFlex)[1] and Parser (JavaCup[2]). The most interesting com-

[1] Available at http://jflex.de/

[2] Available at http://www2.cs.tum.edu/projects/cup/

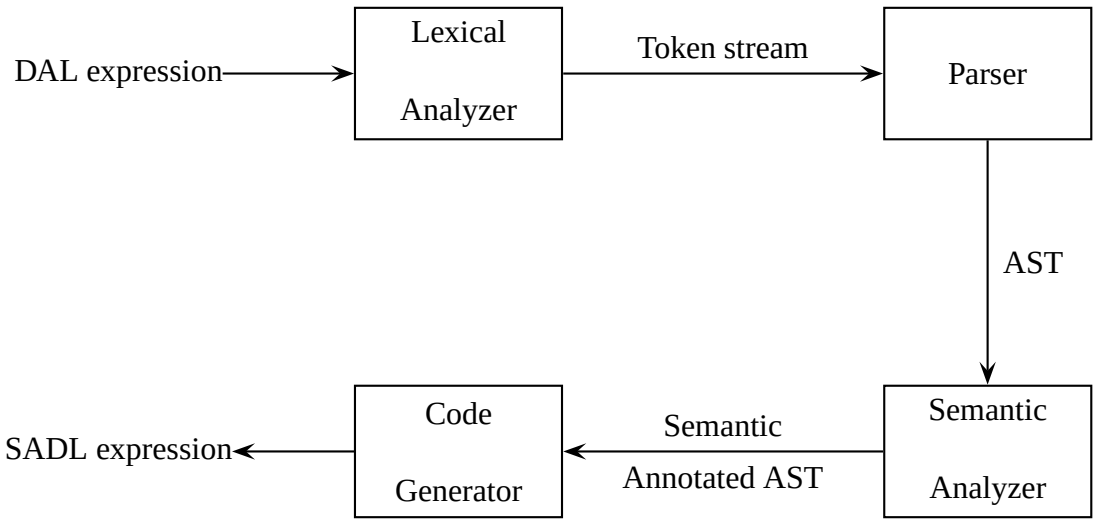

Figure 6.1: DAL Translator processing pipeline

ponents are the Semantic Analyzer and the Code Generator. Both of these components employ domain knowledge to help disambiguate concepts. For example, the Semantic Analyzer uses the transitive `cead:castable` ontological relation to resolve ambiguities when processing overloaded concepts, e.g. operators, action concepts as presented in sections 6.1.3 and 6.1.5.

The DALTranslator is designed using Visitor design pattern (Gamma, Helm, Johnson & Vlissides 1995) so that for each type of AST node, the DALTranslator has a corresponding method to process the code generation for that AST node. DALTranslator is organized as three visitors:

TopDeclVisitor : is our Semantic Analyzer. It is the top-level visitor for processing AST's declaration nodes, such as Variable Declaration Nodes. During this process, it is

assigned correct types (linked to conceptURI) for all declared variables including input, output and local variables. The information for this linking process is provided mostly by the user dictionary. This type information will be used later by ConceptGeneratorVisitor to generate DDVM instructions, especially for selecting correct overloaded operators like $+, -, *, /$, etc.

ConceptGeneratorVisitor : implements our Code Generator. It is the main visitor which is responsible for generating DDVM instructions from AST. There are some cases involving some LHS nodes of an assignment that need to be handled by another visitor LHSVisitor.

LHSVisitor is responsible for generating code for LHS of an assignment. LHSVisitor uses ConceptGeneratorVisitor as its ValueVisitor as specified in some of its methods.

6.1 ConceptGeneratorVisitor

The ConceptGeneratorVisitor starts with the first command node in the body of a DAL algorithm (ignoring top declaration nodes, which are processed by the TopDeclVisitor). In this section, I discuss the process of handling each node in some detail with the following pattern:

1. Present the BNF rules that creates the AST node.
2. List the properties of that AST node, such as list of its children, LHS term and LHS term.
3. Show the algorithm to generate the DDVM instructions for that particular node.

6.1.1 Literals

The LiteralNode is generated by the following BNF rules:

```
1 literal ::= STR_VALUE:v | INT_VALUE:v | FLOAT_VALUE:v
```

From this rule the LiteralNode is created with properties `value`=v, and the type is determined by the type of the literal. The algorithm for generating DDVM instructions is shown in Algorithm 6.1. The `EMITCONSTANTLOAD(v, t)` procedure generates the following

Algorithm 6.1 Generating literal load algorithm

```
1: procedure VISIT(n: LiteralNode)                ▷ Visiting a literal node
2:     EMITCONSTANTLOAD(n.value, n.type)
3: end procedure
```

DDVM instructions.

```
1 <pushStr value="v" />
2 <loadConst conceptURI="URI(t)" />
```

Note that the result of this loadConst will be stored on TOS.

6.1.2 Local Reference

The LocalReferenceNode is created by the following BNF rules:

```
1 term ::= IDENTIFIER:id
```

From this rule, the LocalReferenceNode gets its property `variable`=id. The algorithm for generating DDVM instructions is shown in Algorithm 6.2 The `EMITLOAD(r)` proce-

Algorithm 6.2 Generating local reference algorithm

1: **procedure** VISIT(n: LocalReferenceNode) ▷ Visiting a local reference node
2: $varAttr \leftarrow currentSymbolTable$.RETRIEVESYMBOL($n$.variable)
3: n.SETRESULTLOCAL($varAttr$.localIndex)
4: EMITLOAD(n.localIndex)
5: **end procedure**

dure generates the DDVM instructions for loading the corresponding register which holds the variable to TOS.

```
1 <load index="r" />
```

6.1.3 Computing Expressions

Computing expressions are generated by the following BNF rules:

```
1 term ::= term:l bin_op term:r | "-" term:l | "not" term:l
2 bin_op ::= "+" | "-" | "*" | "/" | "%"
3            | "==" | "!=" | "<" | ">"
4            | "<=" | ">="
5            | "and" | "or"
```

The algorithm for generating DDVM instructions is shown in Algorithm 6.3. The EMIT-OPERATION(n) procedure generates the DDVM instructions for the arithmetic operations.

Algorithm 6.3 Generating computing expression algorithm

1: **procedure** VISIT(n: Computing) ▷ Visiting a computing node such as AddNode, SubtractNode, etc.
2: VISITCHILDREN(n)
3: $loc \leftarrow$ ALLOCLOCAL( )
4: n.SETRESULTLOCAL(loc)
5: EMITOPERATION(n)
6: **end procedure**

The procedure finds the correct concept linked with the operation with a matching signature in the ontology. For example, the operation $+$ could be the addition for integers

`http://bula1.cs.uiowa.edu/owl/arithmetic.owl#addI`

or the addition of real numbers

`http://bula1.cs.uiowa.edu/owl/arithmetic.owl#addR`

The instruction generator will find the first concept whose signature matches the input parameters type. An input parameter type is considered as matching the concept signature's parameter type if the input parameter type is castable to the concept signature's parameter type.

For example, in the expression $1 + 3.1$, DDVM will generate $+$ as `addR`. Because 3.1 is of type `#Real`, the concept `#addI` doesn't match. However, there exists a filter `#intToDouble` which converts an `#Integer` number to a #Real number. Thus, type #Integer is castable to type `#Real`. Therefore, the concept `#addR` matches. So, we get

```
1 <addR xmlns="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#"
    params="2" />
```

6.1.4 Assignment

The assignment node is generated by the following BNF rule:

```
1 assign_cmd ::= term:lhs "=" term:t ";"
```

So, its `lhs` and `rhs` properties are pointing to its children lhs and t respectively. The algorithm for generating DDVM instructions is shown in Algorithm 6.4.

Algorithm 6.4 Generating assignment algorithm

1: **procedure** VISIT(n: AssignNode) ▷ Visiting an assignment node.
2: $lhsVisitor \leftarrow$ **new** LHSVISITOR(**this**)
3: $lhsVisitor$.VISIT(n.lhs)
4: VISIT(n.rhs)
5: $lhsVisitor$.EMITSTORE(n.rhs.GETRESULTLOCAL())
6: **end procedure**

The $lhsVisitor$.EMITSTORE procedure will be discussed in Section 6.2.

6.1.5 Phrase Node

PhraseNodes are generated by the following grammar rules:

```
1 term_list ::= term:p | term_list:l COMMA term:p
2 phrase ::= IDENTIFIER:id "(" term_list:vl ")"
3 phrase ::= IDENTIFIER:id "(" ")"
```

The algorithm for its code generation is shown in Algorithm 6.5. The FINDMATCHEDSIG-

Algorithm 6.5 Generating phrases algorithm

1: **procedure** VISIT(n: PhraseNode) ▷ Visiting a phrase node.

2: $conceptURI \leftarrow$FINDMATCHEDSIGNATURE(n.concept)

3: VISITCHILDREN(n)

4: EMITACTIONCONCEPT($conceptURI$)

5: **end procedure**

NATURE procedure is the same as that of Algorithm 6.3. The `EMITACTIONCONCEPT` procedure emits DDVM instructions for domain action concept. For example, with the expression `sqrt(4.0)`, the concept generator will first look up a matched concept from user dictionary. Assume that it finds `http://bula1.cs.uiowa.edu/owl/arithmetic.owl\#sqrtR`. The concept generator then finds the number of input parameters that the concept `sqrt` has and gets 1. So, the generator emits the following DDVM instruction:

```
1 <sqrtR xmlns="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#"
    params="1" />
```

6.1.6 Field Reference

FieldReferenceNode n is generated by the following BNF rule:

```
1 term ::= term:t "." IDENTIFIER:id
```

The properties are assigned as follows: n.instance = t, n.fieldName = id. The algorithm for generating DDVM instructions is shown in Algorithm 6.6. The `EMITFIELDREFERENCE(n)`

Algorithm 6.6 Generating field reference algorithm

```
1: procedure VISIT(n: FieldReferenceNode)        ▷ Visiting a field reference node.
2:     VISIT(n.instance)
3:     EMITFIELDREFERENCE(n)
4: end procedure
```

procedure will generate the DDVM `getfield` instruction with the field name obtained from the FieldReferenceNode n.

```
1   <getfield field="fieldName" />
```

6.1.7 Array Reference

The ArrayReferenceNode n is created by the BNF rule:

```
1 term ::= term:t "[" term:i "]"
```

So, from this rule, we have n.array = t, n.index = i. The corresponding algorithm is shown in Algorithm 6.7. The `EMITARRAYREFERENCE(`n`)` procedure generates the DDVM

Algorithm 6.7 Generating array reference algorithm

```
1: procedure VISIT(n: ArrayReferenceNode)        ▷ Visiting a array reference node.
2:     VISIT(n.GETARRAY())
3:     VISIT(n.GETINDEX())
4:     EMITARRAYREFERENCE(n)
5: end procedure
```

`aload` instruction. The `type` attribute required by the `aload` instruction is the base type of this array. This base type is retrieved from the ontology via the property `cead:hasBaseType` of the class `cead:ArrayDataConcept`.

```
1  <aload type="BASETYPE(n.GETARRAY())" />
```

6.1.8 Conditional Branching

The IfNode n is generated by the following grammar rules:

```
1 if_cmd ::= "if" term:p "then"  cmd_list:cl "endif" ";"
2 if_cmd ::= "if" term:p "then"  cmd_list:cl "else" cmd_list:cl2 "
     endif" ";"
```

Properties for n are specified as follows: n.boolExpr = p, n.action = cl, n.alternativeAction = cl2. Thus, the algorithm for generating DDVM instructions is shown in Algorithm 6.8. In this algorithm, the procedure `GENLABEL` generates a new label for $falseLabel$ and $endLabel$. `EMITJUMPFALSE` will produce

```
1 <jumpfalse label="falseLabel" />
```

Algorithm 6.8 Generating conditional branching algorithm

```
1:  procedure VISIT(n: IfNode)                    ▷ Visiting an IF-THEN node.
2:      falseLabel ←GENLABEL( )
3:      endLabel ←GENLABEL( )
4:      VISIT(n.GETBOOLEXPRESSION())
5:      EMITJUMPFALSE(falseLabel)
6:      VISIT(n.GETACTION())
7:      EMITJUMP(endLabel)
8:      EMITLABEL(falseLabel)
9:      if n.alternativeAction is not empty then
10:         VISIT(n.GETALTERNATIVEACTION())
11:     end if
12:     EMITLABEL(endLabel)
13: end procedure
```

while `EMITJUMP` generates

```
1 <jump label="endLabel" />
```

So in total, this algorithm generates the following DDVM instructions template:

```
1 <!-- check boolExpr code -->
2 <jumpfalse label="falseLabel" />
3 <!-- action code -->
4 <jump label="endLabel" />
5 <label name="falseLabel" />
6 <!-- alternativeAction code -->
7 <label name="endLabel" />
```

6.1.9 Loops

The WhileNode n is generated by the BNF rule:

```
1 while_cmd ::= "while" term:p "do" cmd_list:cl "endwhile" ";"
```

Its properties are set up as n.boolExpr = p, n.action = cl. With these information, the code generation algorithm is shown in Algorithm 6.9. Procedures in this algorithm are very similar to those of Algorithm 6.8. However, the code template is different, as seen below:

```
1 <label name="loopLabel" />
2 <!-- check boolExpr code -->
3 <jumpfalse label="doneLabel" />
4 <!-- action code -->
5 <jump label="loopLabel" />
6 <label name="doneLabel" />
```

An actual code sample for while-loop is shown in the SADL code for the Euclidean algorithm in Listing D.1.

Algorithm 6.9 Generating loops algorithm

```
1: procedure VISIT(n: WhileNode)                    ▷ Visiting a WHILE node.
2:     doneLabel ←GENLABEL( )
3:     loopLabel ←GENLABEL( )
4:     EMITLABEL(loopLabel)
5:     VISIT(n.GETBOOLEXPRESSION())
6:     EMITJUMPFALSE(doneLabel)
7:     VISIT(n.GETACTION())
8:     EMITJUMP(loopLabel)
9:     EMITLABEL(doneLabel)
10: end procedure
```

6.2 LHSVisitor

6.2.1 Local References

The `LocalStore` command, when executed by the `EMITSTORE` procedure from Section 6.1.4, will emit the DDVM `store` instruction. The information about the register to be used is the localIndex of the variable.

```
1 <store index="localIndex" />
```

6.2.2 Field Reference

The `FieldStore` command, when executed by the `EMITSTORE` procedure from Section 6.1.4, will emit the DDVM `putfield` instruction. The information about the

Algorithm 6.10 Generating LHS local reference algorithm

1: **procedure** VISIT(n: LocalReferenceNode) ▷ Visiting a local reference node

2: $varAttr \leftarrow currentSymbolTable$.RETRIEVESYMBOL($n$.variable)

3: n.SETTYPE($varAttr$.variableType)

4: SETSTORE(**new** LocalStore(n.GETTYPE(), $varAttr$.localIndex)))

5: **end procedure**

Algorithm 6.11 Generating LHS field reference algorithm

1: **procedure** VISIT(n: FieldReferenceNode) ▷ Visiting a field reference node

2: $valueVisitor$.VISIT(n.instance)

3: SETSTORE(**new** FieldStore(n.GETTYPE(), n.fieldName)))

4: **end procedure**

field type (ft) and the field name (fn) is provided by the FieldStore object.

```
1  <putfield type="ft" field="fn" />
```

6.2.3 Array Reference

The `ArrayStore` command, when executed by the `EMITSTORE` procedure from Section 6.1.4, will emit the DDVM `astore` instruction. The information about the field type (ft) is provided by the FieldStore object.

```
1  <astore type="ft" />
```

Algorithm 6.12 Generating LHS array reference algorithm

```
1: procedure VISIT(n: ArrayReferenceNode)          ▷ Visiting a array reference node
2:     valueVisitor.VISIT(n.GETARRAY())
3:     valueVisitor.VISIT(n.GETINDEX())
4:     SETSTORE(new ArrayStore(n.GETARRAY().GETTYPE()))
5: end procedure
```

CHAPTER 7
DOMAIN ONTOLOGY EVOLUTION

One of the key ideas in our approach is to provide a mechanism that allows domain experts to create and extend their domain knowledge base. The process of expanding a user's knowledge base via DAL expressions is called Domain Ontology Evolution (DOE). This process simulates and records the process of a domain expert learning about a domain and keeps expanding her knowledge base during the problem solving process. For example, during a problem solving process, a domain expert can discover a new action concept which is represented as a DAL algorithm such as the steps leading to the solution of a quadratic equation ($ax^2+bx+c=0$). Or, she might discover a new data concept such as the complex numbers when she tries to solve a quadratic equation which has no real number solutions (the case when $\Delta = b^2 - 4ac < 0$). The approach of this thesis to addressing these two cases is implemented by two procedures `add2Onto` and `addData2Onto` presented in the following sections.

7.1 Creating new Action Concepts - add2Onto

The general idea for adding a new action concept to the user ontology consists of the following steps:

1. generate a web service instance for that action concept from the SADL code of that concept,
2. create a new individual of class `ActionConcept` in user own ontology (UOO),
3. automatically perform the process of associating the new concept with the generated

service instance.

The most important design idea is step 1, i.e. to export the action concept as a web service instead of exposing the SADL code or DAL solution directly. If the process of evolution requires the SADL code or DAL solution to be exposed directly, it implies a cascading exposure of all other SADL codes of concepts that the original new concept imported. Moreover, the execution of these SADL codes will take place on the user machine, which will involve the problem of having no access to certain resources that these SADL codes require. On the other hand, if we export the action concept as a web service, the concept will be exposed as a standalone and composable concept like every other primitive concept in the domain ontology.

This idea is formally expressed in Algorithm 7.1. The GENERATESADL procedure

Algorithm 7.1 Creating new action concept algorithm

```
1: procedure ADD2ONTO(DAL: DALExpression)        ▷ create new action concept.
2:     GENERATESADL(DAL)
3:     ast ← PARSE(DAL)
4:     GENERATEXSD(ast)
5:     GENERATEWSDL(ast)
6:     CREATENEWACTIONCONCEPT(ast)
7: end procedure
```

uses a DALTranslator discussed in chapter 6 to generate the SADL expression from the

DAL expression. This SADL expression is then stored in the user private space for later use as the service instance of this DAL action concept. Next we obtain an AST of the DAL expression from the `PARSE` procedure.

The `GENERATEXSD` procedure walks through the ast tree to generate the corresponding XSD schema file for the DAL service instance from the input and output declarations of the DAL expression.

The `GENERATEWSDL` procedure also walks the ast to generate the WSDL file for the DAL service instance. The patterns for generating the WSDL file are shown in Table 7.1.

Finally, the `CREATEACTIONNEWCONCEPT` procedure, shown in Algorithm 7.2, creates a new action concept individual $acInd$ in the user own ontology. In this procedure, for each input node from the ast, one `Input` individual is created with corresponding order and type, then assigned to the property `hasInput` of $acInv$. Similar steps are repeated for output parameter.

The above scenario is demonstrated further with the example in high school algebra that maps the algorithm solving quadratic equations into a new concept called `Solver`. We assume that the DAL expression of the algorithm that solves quadratic equations is written as follows and saved as the file `solver.dal` shown in Listing 7.1.

Then using the DAL Console program, the user executes the command

```
1 add2Onto solver.dal
```

The generated SADL file is shown in Listing 7.2.

WSDL file Properties	*Value*	*Comment*
schemaLocation	the URI of XSD file generated by `GENERATEXSD`	
input message	(conceptName)	XSD type is named as conceptName
output message	(conceptName)Response	XSD type for output message of concept `add` is like addResponse
portType	(conceptName)PortType	for example `addPortType`
operation	(conceptName)	for example `add`
service	(conceptName)	name of the web service instance is after the concept name
location	http://(server)/OntologyManager/services/(conceptName)	URI of the web service instance

Table 7.1: Patterns for generating WSDL files

Algorithm 7.2 Creating new action concept individual algorithm

1: **procedure** CREATEACTIONNEWCONCEPT(ast: ProgramNode) ▷ create new action concept.

2: $m \leftarrow$ LOADONTOLOGYMODEL(user-CEAD.owl)

3: $acInd \leftarrow m$.CREATEINDIVIDUAL(ast.ConceptName, "ActionConcept")

4: **for** $i = 1$ to ast.InputList.SIZE **do**

5: $inputNode \leftarrow ast$.InputList.GET(i)

6: $inputInd \leftarrow m$.CREATEINDIVIDUAL(ast.ConceptName+i, "Input")

7: $inputInd$.ADDPROPERTY("inputType", $inputNode$.GETTYPE())

8: $inputInd$.ADDPROPERTY("order", i)

9: $acInd$.ADDPROPERTY("hasInput", $inInd$)

10: **end for**

11: $acInd$.ADDPROPERTY("hasOutput", ast.GETOUTPUTTYPE())

12: $serviceInd \leftarrow m$.CREATEINDIVIDUAL(ast.ConceptName + "ServiceInstance", "ServiceInstance")

13: SETUPWSDLPROPERTY($serviceInd$) ▷ Assign service information from WSDL to this individual

14: $agentInd \leftarrow m$.CREATEINDIVIDUAL(ast.ConceptName + "Agent", "Agent")

15: $acInd$.ADDPROPERTY("hasAgent", $agentInd$)

16: $agentInd$.ADDPROPERTY("implementedBy", $serviceInd$)

17: **end procedure**

OWL Properties	*Value*	*Comment*
Input		
schemaLocation	the URI of XSD file generated by `GENERATEXSD`	
input message	(conceptName)	XSD type is named as conceptName
output message	(conceptName)Response	XSD type for output message of concept `add` is like addResponse
portType	(conceptName)PortType	for example `addPortType`
operation	(conceptName)	for example `add`
service	(conceptName)	name of the web service instance is after the concept name
location	http://(server)/OntologyManager/services/(conceptName)	URI of the web service instance

Table 7.2: Patterns for generating OWL individual

```
concept: "Solver";
description: "This is an equation solver.";
message input: "Provide the coeffs of your quadratic equation ax
   ^2 + bx + c = 0";
input: a : real, b : real, c : real;
output: result: RealPair;
local: t : real, x1: real, x2: real;
t = b * b - 4 * a * c;
if t > 0 then
   x1 = (-b - sqrt(t)) / (2 * a);
   x2 = (-b + sqrt(t)) / (2 * a);
   result.first = x1;
   result.second = x2;
else
   print("the equation has no real solution");
endif;
```

Listing 7.1: DAL algorithm for solving quadratic equations

The OWL entry for the generated `Solver` concept is shown in Lising 7.3. From now on, the user can use the concept "Solver" as any other primitive concepts by executing the command

```
>Solver(1, 4, 3)
(first: -3.0, second: -1.0, )
```

or she can use this concept in another DAL expression for example

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

Notice that since all the code involved in the `ad2onto` procedure is automatically created the student learning to solve quadratic equations using the DALSystem manipulates only algebraic concepts.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <sadl xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <declaration>
4      <imports>
5        <importOntology uri="URI(arithmeticCEAD.owl)" />
6        <importOntology uri="URI(cead.owl)" />
7        <importOntology uri="URI(cuongbk-CEAD.owl)" />
8      </imports>
9      <inputs>
10       <input type="ari:Real" index="4" />
11       <input type="ari:Real" index="5" />
12       <input type="ari:Real" index="6" />
13     </inputs>
14     <outputs>
15       <output type="ari:RealPair" index="7" />
16     </outputs>
17   </declaration>
18   <init type="ari:Real" index="1" />
19   <init type="ari:Real" index="2" />
20   <init type="ari:Real" index="3" />
21   <init type="ari:RealPair" index="7" />
22   <load index="5" />
23   <load index="5" />
24   <multiplyR xmlns="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
       " params="2" />
25   ...
26   <load index="7" />
27   <load index="2" />
28   <putfield type="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
       Real" field="first" />
29   <load index="7" />
30   <load index="3" />
31   <putfield type="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
       Real" field="second" />
32   <jump label="label2" />
33   <label name="label1" />
34   <pushStr value="the equation has no real solution" />
35   <loadConst conceptURI="http://bula1.cs.uiowa.edu/owl/cead.owl#
       String" />
36   <printTOS />
37   <label name="label2" />
38 </sadl>
```

Listing 7.2: The generated SADL file for Solver concept.

```
<cead:ActionConcept rdf:about="cuongbk:Solver">
    <cead:hasAgent>
      <cead:Agent rdf:about="cuongbk:SolverAgent">
        <cead:implementedBy>
          <cead:ServiceInstance rdf:about="
              cuongbk:SolverServiceInstance">
            <cead:wsdlFile>http://localhost:8080/OntologyManager/
              resources/sadl/Solver.wsdl</cead:wsdlFile>
            <cead:serviceName>Solver</cead:serviceName>
            <cead:operationName>Solver</cead:operationName>
            <cead:portName>SolverHttpSoap11Endpoint</
              cead:portName>
          </cead:ServiceInstance>
        </cead:implementedBy>
      </cead:Agent>
    </cead:hasAgent>
    <cead:hasOutput rdf:resource="ari:RealPair" />
    <cead:hasInput>
      <cead:Input rdf:about="cuongbk:SolverInput3">
        <cead:inputType rdf:resource="ari:Real"/>
        <cead:inputName>c</cead:inputName>
        <cead:order>3</cead:order>
      </cead:Input>
    </cead:hasInput>
    <cead:hasInput>
      <cead:Input rdf:about="cuongbk:SolverInput2">
        <cead:inputType rdf:resource="ari:Real"/>
        <cead:inputName>b</cead:inputName>
        <cead:order>2</cead:order>
      </cead:Input>
    </cead:hasInput>
    <cead:hasInput>
      <cead:Input rdf:about="cuongbk:SolverInput1">
        <cead:inputType rdf:resource="ari:Real"/>
        <cead:inputName>a</cead:inputName>
        <cead:order>1</cead:order>
      </cead:Input>
    </cead:hasInput>
    <cead:inputMessage>Provide the coeffs of your quadratic
       equation ax^2 + bx + c = 0</cead:inputMessage>
    <cead:description>This is an equation solver.</
       cead:description>
  </cead:ActionConcept>
```

Listing 7.3: OWL entry for the Solver concept.

7.2 Creating new Data Concepts - addData2Onto

The general idea of the algorithm for creating data concepts from DAL expression is:

1. create the corresponding `DataConcept` individual for the domain concept declared in the DAL. For example, the `record` type in DAL is mapped to `ComposedDataConcept` in CEAD Ontology, the `array` type of DAL is mapped to either `UnconstrainedArray` or `ConstrainedArray` depending on whether the range is specified in the DAL expression.
2. generate the corresponding XSD type for the `DataConcept`. The generated XSD type is a composition of all the XSD types of subcomponents.
3. link automatically the URI of the generated XSD with the `DataConcept` individual property `dataType`.

The following sections discusses in more details each case of data concepts supported by the DALSystem.

7.2.1 Creating composed data concepts

The DAL syntax for creating a composed data concept is:

```
1  concept: "<concept-name>";
2  description: "<concept-description>";
3  local: concept <concept-name> is
4    record
5       field1 : type1;
6       field2 : type2;
7       ...
8       fieldN : typeN;
9    endrecord;
10 endconcept;
```

Where <concept-name> is the name she assigns to the composed concept, field$X$ and type$X$, $X = 1 \ldots n$ are the name and the type of the field X in this concept. The <concept-description> part will be used to display to the domain user when she queries information about this concept, for example, when the domain expert discovers the new data concept Complex after she cannot find the solution for her quadratic equation with real numbers. Each Complex number has two real number components real part and imaginary part. She can write a new data concept Complex in a file named `complex.dal` as follows:

```
concept: "Complex";
description: "Complex concept in complex analysis.";
local: concept Complex is
  record
     ImgPart : real;
     RealPart : real;
  endrecord;
endconcept;
```

When a DAL expression for a composed data concept is sent to the OntologyManager from the DALConsole via the command `addData2Onto`, the OntologyManager will create an `ComposedDataConcept` individual c with the ID set to the name of the concept. For each field in the concept description, the OntologyManager generates a `Field` individual for it with the corresponding properties `hasName` and `hasType`. These `Field` individuals are then added to the individual c under the property `hasField`. For example, the Complex data concept is generated as follows:

```
<cead:ComposedDataConcept rdf:about="#Complex">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
    <cead:description>This is the complex number concept in the arithmetics domain.
    </cead:description>
    <cead:hasField>
        <cead:Field rdf:ID="ComplexField1">
        <cead:hasType rdf:resource="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#Real"/>
        <cead:hasName>ImgPart</cead:hasName>
        </cead:Field>
    </cead:hasField>
    <cead:hasField>
        <cead:Field rdf:ID="ComplexField2">
        <cead:hasType rdf:resource="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#Real"/>
        <cead:hasName>RealPart</cead:hasName>
        </cead:Field>
    </cead:hasField>
    <cead:dataType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        complex:Complex
    </cead:dataType>
</cead:ComposedDataConcept>
```

where XSD type `complex:Complex` is defined as follows:

```
<xs:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="complex">
    <xs:complexType name="Complex">
        <xs:sequence>
            <xs:element minOccurs="0" name="RealPart" type="xs:double"/>
            <xs:element minOccurs="0" name="ImgPart" type="xs:double"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

After the concept is added to the ontology, it can be used in DAL expressions as other primitive data concepts are used. Moreover, to access any field of a variable x of a composed data concept type, the user uses the syntax: x.fieldName. For example, if x is a variable of type Complex, to access its RealPart field, the user writes x.RealPart. A more complex example is shown in the following listing with SolverC that solves quadratic equations that have complex solutions:

```
1  concept: "SolverC";
2  description: "This is an quadratic equation solver with complex
       solution.";
3
4  input: a : real, b : real, c : real;
5  output: result: ComplexPair;
6  local: t : real, u : real, x1 : Complex, x2 : Complex;
7
8  t = b * b - 4.0 * a * c;
9  print(t);
10 if t > 0.0 then
11    x1 = constructC((-b + sqrt(t)) / (2.0 * a), 0.0);
12    x2 = constructC((-b - sqrt(t)) / (2.0 * a), 0.0);
13 else
14    u = sqrt(-t) / (2.0 * a);
15    x1 = constructC(-b / (2.0 * a), u);
16    x2 = constructC(-b / (2.0 * a), -u);
17 endif;
18 result.first = x1;
19 result.second = x2;
```

where (1) ComplexPair is also a composed data concept with the two Complex fields: `first` and `second`; (2) constructC is an auxiliary action concept for constructing a complex number written as follows:

```
1 concept: "constructC";
2 description: "construct a complex number from two real numbers,
     the first number is the real part, the second number is the
     imaginary part.";
3 message input: "input message";
4 input: x: real, y: real;
5 output: c: Complex;
6 c.RealPart = x;
7 c.ImgPart = y;
```

Again, the XML code is automatically generated, mimicking the process of a student learning to solve quadratic equations.

7.2.2 Creating array data concepts

There are two types of arrays in DAL: Unconstrained array data concept and constrained array data concept. A constrained array data concept is wanted if there is a range for lowerbound and upperbound of the array. Otherwise it is an unconstrained array data concept. For example, `String` could be considered as an unconstrained array data concepts of `Character`. But, a vector of 3-dimensional space is a constrained array data concept.

The DAL syntax for creating a constrained array data concept is:

```
1 concept: "<concept-name>";
2 description: "<concept-description>";
3 local: concept <concept-name> is
4     array (lowerbound .. upperbound) of <base-type>
5 endconcept;
```

The DAL syntax for creating unconstrained array data concept is:

```
concept: "<concept-name>";
description: "<concept-description>";
local: concept <concept-name> is
    array of <base-type>;
endconcept;
```

In these definition schemes the <concept-name> is the user name of the array concept, <base-type> is the type of each element in the array, and <concept-description> is similar to that of `ComposedDataConcept`.

For example, when the domain expert wants to have the new data concept Vector of 3-dimensional space to work with linear equations, she can specify it in a file called vector3d.dal as follows:

```
concept: "Vector3D";
description: "3-dimensional space vector";
local: concept Vector3D is
    array (1 .. 3) of real;
endconcept;
```

After the user sends this file to the OntologyManager via the command `addData2Onto`, the following `ArrayDataConcept` individual is generated in her own ontology:

```
<cead:ConstrainedArray rdf:about="http://localhost:8080/cuongbk.
   owl#Vector3D">
    <cead:hasBaseType rdf:resource="http://bula1.cs.uiowa.edu/owl
       /arithmetic.owl#Real"/>
    <cead:hasUpperBound>3</cead:hasUpperBound>
    <cead:hasLowerBound>1</cead:hasLowerBound>
    <cead:description>3-dimensional space vector</
       cead:description>
    <cead:dataType>ari:Vector3D</cead:dataType>
</cead:ConstrainedArray>
```

For an unconstrained version of the concept vector, she would write:

```
concept: "Vector";
description: "n–dimensional space vector";
local: concept Vector is
    array of real;
endconcept;
```

and the generated individual should be:

```
<cead:UnconstrainedArray rdf:about="http://localhost:8080/cuongbk
    .owl#Vector">
    <cead:hasBaseType rdf:resource="http://bula1.cs.uiowa.edu/owl
        /arithmetic.owl#Real"/>
    <cead:description>n–dimensional space vector</
        cead:description>
    <cead:dataType>ari:Vector</cead:dataType>
</cead:UnconstrainedArray>
```

After adding these concept definitions to the ontology, the domain expert can write DAL algorithms which use these concepts as she would do using primitive concepts. To access the i-th element of an array x, the user writes: $x[i]$. For example, the scalar product of two Vector3D can be written as:

```
concept: "product3D";
description: "compute scalar product of two 3D vectors.";
input: v1: Vector3D, v2: Vector3D;
output: p: real;
p = (v1[1]) * (v2[1]) + (v1[2]) * (v2[2]) + (v1[3]) * (v2[3]);
```

Finally, n-dimensional vectors can be used in solving linear equation system using HouseHolder reduction method as shown in Appendix C.

CHAPTER 8
DALSYSTEM

This chapter provides some implementation details about of the actual DALSystem. First, I discuss the deployment details of the DALSystem architecture previously shown in Figure 1.2, then SADLServlet is introduced, and finally the implementation of DDVM is sketched. The DALSystem is developed using Java and runs on a UNIX server (at the server `bula1.cs.uiowa.edu`) at the Department of Computer Science at University of Iowa. The library for ontology manipulation and reasoning is the Apache Jena 2.6.4[1].

8.1 DALSystem Deployment

The components of a DALSystem for an application domain are shown in Figure 8.1. The components of this diagram are deployed on two web servers.

DAL Console: allows the user to interact with logical concepts in her ontology space including her private and shared ontologies.

DAL Translator: translates user DAL expressions into intermediate language called SADL.

DDVM: receives the SADL expression, executes it and returns result to the caller.

Ontology Manager: is responsible for managing user private ontology such as adding new concepts and removing concepts from user's private ontology. It also provides the look up service for other components like DAL Translator and DDVM.

SADL Servlet: serves as a wrapper around DDVM component so that the outside world can interact with user concepts like normal web services via SOAP protocol.

[1] Available at http://jena.apache.org/

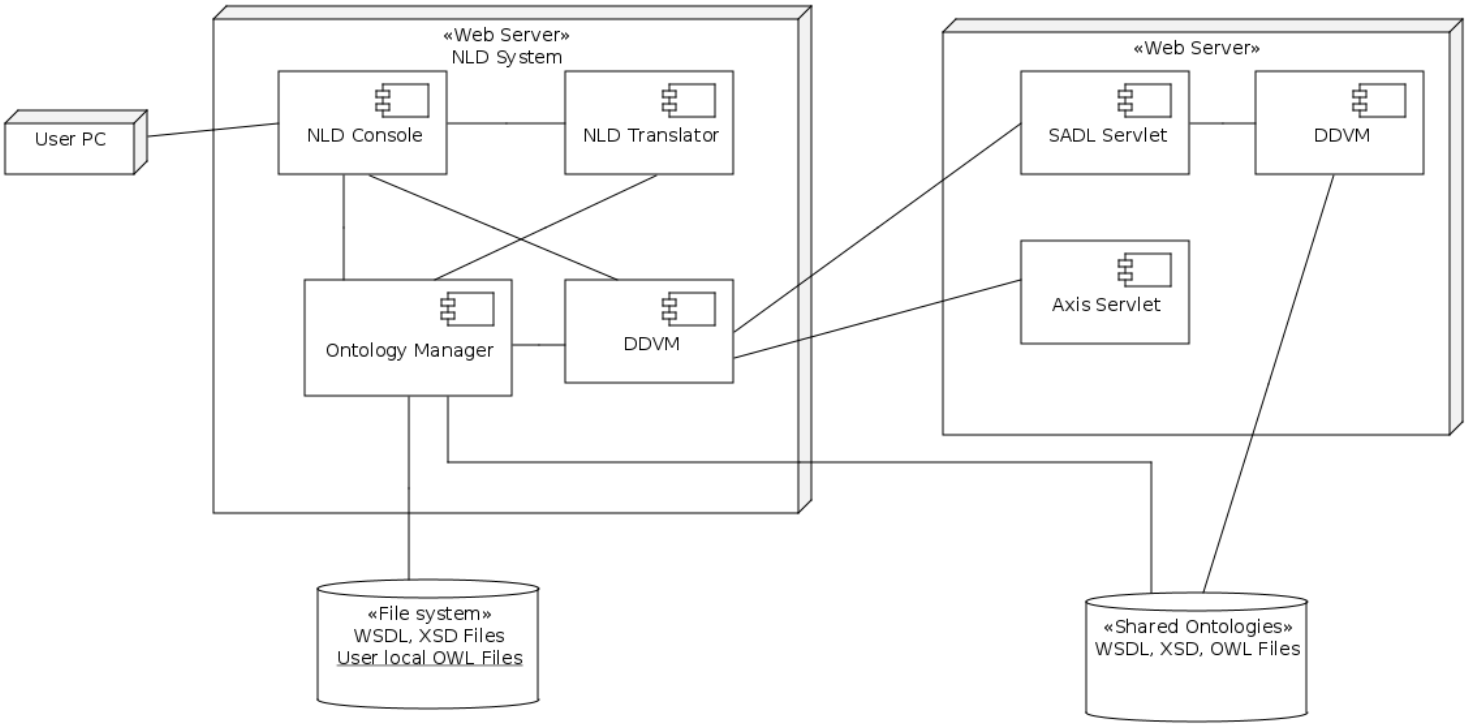

Figure 8.1: DALSystem components deployment

The components contained in the web server on the left hand side of the component diagram form a complete DALSystem in the user private space where their interactions have already been discussed in section 1.4 The components depicted in the Web server on the right hand side show how a user concept can be reused like a normal web service. When the user adds a new concept into her ontologies (private or shared ones), the Ontology Manager translates her DAL expression to SADL expression and stored in her private space. The URI for accessing the SADL expression is then published as a web service via a WSDL file. This SADL expression can then be accessed for execution by another DDVM via the SADL Servlet as presented in section 8.2.

Even though a DALSystem can be deployed to a local network, it is recommended to be deployed to a cloud computing environment where high speed Internet connection and on-demand computing resource scalability are provided. These are ideal conditions for virtual machines operating on a network like our DDVM. We assume that the cloud computing environment which hosts the DALSystem would have an administration system to manage user accounts and use a subscription model for operation as suggested in (Rus 2013). The user administration system allows various users to register for the DALSystem use on a given problem domain. After registering for an account, each user is granted access to domain expert ontologies (DEO) and all computational artifacts associated with concepts in these ontologies. A private user space is also provided for the user to store her own ontology (UOO). The user subscription for a domain D will activate the DALSystem installation procedure with the required domain ontology. After that the user can use the domain concepts from provided ontologies or can evolve the problem domain she subscribed for with

new concepts she learned and/or created during her own problem solving process as discussed in Chapter 7. The DALSystem manager could also offer to buy the knowledge developed by the user and update the domain ontology, thus ensuring domain evolution with new concepts developed by the respective user. When the user decides to cancel her subscription and leave the system, she can also offer to sell her concepts to the DALSystem manager in order to retain these concepts for later use. This evolution model ensures the domain knowledge expanded by knowledge gains from all domain experts during their problem solving processes.

The diagram in Figure 8.2, a slightly modified version of Figure 2 in (Rus 2013), illustrates clearly this cloud implementation of the DALSystem. In this diagram, the Cloud Administrator is represented as the smiling face on the top. There are k domain users at the bottom whose activities are numbered according to their sequence, with the first activity is subscribing to the DALSystem. The next activity is the installation of the DALSystem onto the user's space. Then the user can use her concepts via the DAL Console. The fourth step is to evolve the user own ontology. Finally, the user can publish her concepts to the shared domain expert ontology in the fifth activity.

8.2 SADL Servlet

SADL Servlet is a Java Servlet (Mordani 2009) which is a service wrapper for DDVM so that it helps the DDVM communicate with the outside world via SOAP protocol just like normal web services. The SADL Servlet was designed to allow one DDVM to invoke another DDVM without any special distinction with other SOAP-based web services.

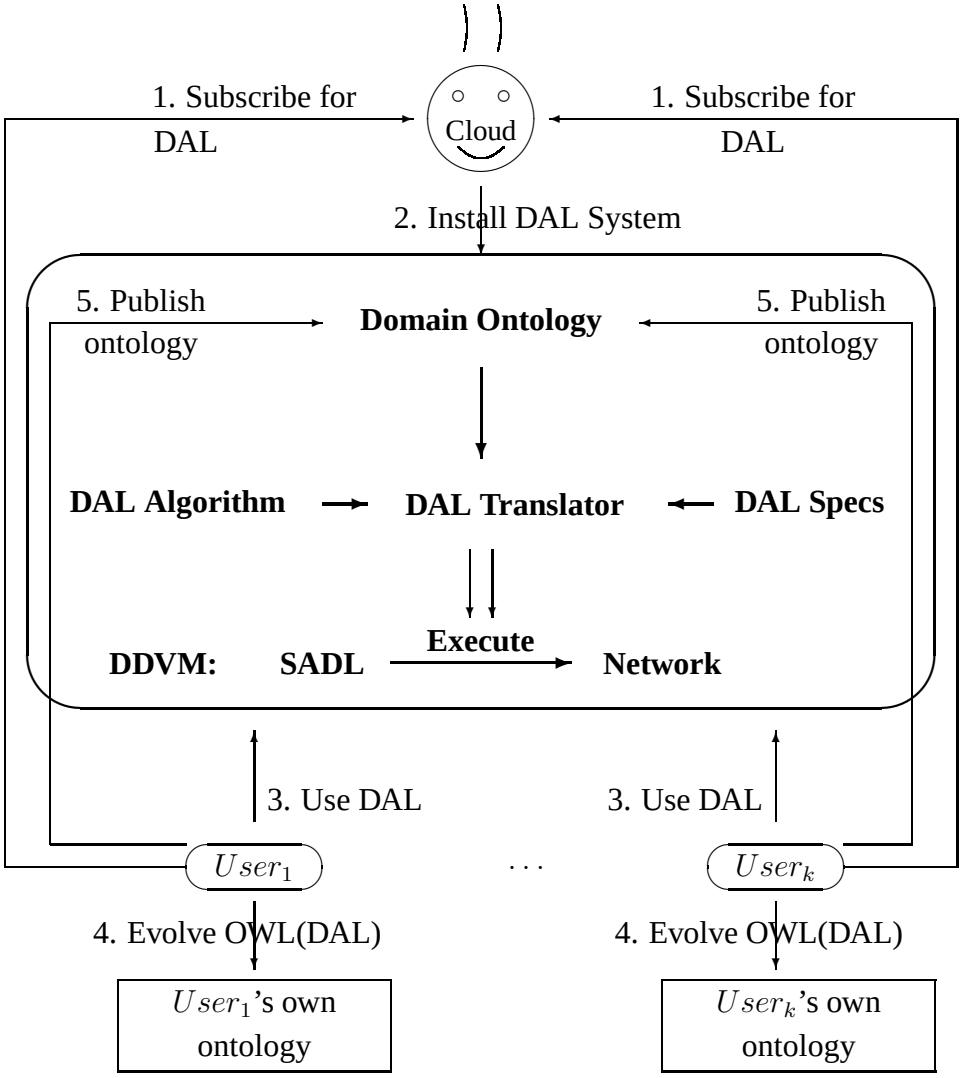

Figure 8.2: Cloud Implementation of the DALSystem

This design also allows a DDVM V_1 to execute remote SADL code without downloading that SADL code to the local DDVM V_1. Thus, we achieve two goals:

1. Improve the security of the SADL code so that the user can confidently share their services without worrying about losing their implementation ideas.
2. The SADL code is executed where it is deployed. That means all the configuration settings and dependencies are in place.

The process of invoking an action concept as a normal web service via the SADL Servlet starts when a DDVM makes a SOAP call to an URI where the SADL Servlet is listening. Based on the request URI, the SADL Servlet retrieves the corresponding SADL expression for the requested service of the action concept. Then, a DDVM is spawned in a new thread by the SADL Servlet and then the DDVM loads the SADL expression into its memory. SADL Servlet analyzes the SOAP request once again to retrieve the input data and transfers it to the DDVM before executing the SADL code. After the execution finishes, the SADL Servlet retrieves the result and returns it back to the initial DDVM as a SOAP response message.

Since the time for creating a DDVM from a SADL file is significant when the ontology is large, it would be inefficient if SADLServlet has to reconstruct a DDVM every time a request comes and destroys the DDVM after that. A solution to make the system more efficient would be to cache the DDVM for each invoked concept. The cache value will be refreshed if the content of the concept is updated, i.e. the SADL code of the concept. Using this technique, the DALSystem reduces the overall execution time by about 3 times.

8.3 Implementation of DDVM

The rationale for virtual machines like DDVM is that they are virtual processors which can provide a common execution platform for one or more DALs in a hardware-independent way. There are several approaches to implement virtual machines (Craig 2006):

- Direct implementation;
- Translation;
- Threaded code.

In this thesis the author chooses to use the direct implementation approach. However, the platform which executes DDVM instructions is a Java Virtual Machine (JVM). We choose JVM to maximize the capability of DDVM to execute on as many hardware platforms as possible without worrying about the variety of hardware processors. In that light, each "computational concept call" instruction (for executing a concept) will be directly executed by a JVM. JVM will make a SOAP call to the remote web service implementing the semantics of the corresponding domain concept. The current execution thread of the DDVM is blocked and waits for the result from the remote server. The remote server will create a process executing the corresponding domain concept executable code (semantics). The result of the call will be pushed back to TOS (Top of Stack) of the local execution scope. The execution thread of the DDVM is notified to continue.

During the execution process, if a domain action concept is encountered, the DDVM will dynamically download the WSDL file of the targeted web service to extract information about parameter names to compose the SOAP message correctly. Therefore, if it has

to download and parse the WSDL every time an action concept is executed, it will be very inefficient. Since the WSDL file hardly changes during the lifetime of a web service deployment, DDVM will only download and parse the WSDL file the first time; it will cache the WSDL parse in its memory until it is destroyed. This caching method speeds up the DDVM and reduce the execution time about 3 times.

CHAPTER 9
CONCLUSIONS

The DALSystem was set out to explore the concept identified in “Liberating Computer User from Programming” (Rus 2008) through a creation of a prototype system. This research has sought to answer two questions:

1. Can we develop a system to be used by a domain expert to integrate computers into her problem solving process?
2. If so, can we demonstrate the system with a particular domain?

This thesis provides affirmative answers to these two questions by the implementation of the DALSystem as a demonstration of integrating computers into the problem solving process of the domain of arithmetic, high-school algebra and vector algebra.

The implementation of the DALSystem for the domain of arithmetic allows domain users to express their computations using domain specific terms and phrases while providing seamless execution of the computation on the network across organization boundaries based on web services composition. The language can be used naturally by domain users because its vocabulary and phrases are characteristic to the domain of arithmetic. The language is also algorithmic because ambiguities of the language arithmeticians use are eliminated by the arithmetic context.

This thesis also proposed a mechanism for the domain user to evolve the domain ontology with new action and data concepts. This mechanism of automating the process of associating computational artifacts with new domain concepts will help domain users

easily expand their ontologies during the problem solving process from a small set of initial primitive concepts.

In this thesis, we also experiemented with the idea of putting data composition information into ontological knowledge base. Such information is vital for the system to provide better input/output handling experience in an interactive mode with domain users.

All the accomplishments discussed above show the potential of the DALSystem. However, there are several problems to be addressed in future research.

The DALSystem is a demonstration of our domain-oriented methodology for the domain of arithmetic. Arithmetic is a well-defined domain which has been studied for thousand years, so it was relatively easy to implement the concept. There are new emerging fields of study that are not very well-defined such as bioinformatics or computational linguistics. Therefore, a better experiment would be an implementation of our methodology for the domain of bioinformatics and/or computational linguistics using Web systems. The incremental nature of the domain ontology makes us believe that breaking the domain thinking inertia is the major problem in front of such experiments.

The DALSystem is currently implemented to work with web services using the SOAP protocol. Even though the SOAP protocol is a well-defined standard, it results in a large overhead for SOAP messages. On the other hand, REST (Erl, Balasubramanian, Carlyle & Pautasso 2012) is a light-weighted protocol without the cost overhead. So implementing a DALSystem to work with the REST protocol is a worthy research direction for the future.

The system implementation suffers from lack of efficiency compared to traditional compiled programs running on local machines. This is mainly because most of the execution times of the experimented concepts are much smaller compared to the communication overhead time. Another reason for lack of efficiency is the sequential implementation of the process of composing web services. After the DDVM sends out a SOAP request, it waits for the response from the server. The execution process can be actually accelerated if the DDVM executes concepts in parallel.

The current implementation of DDVM for web services composition is very simple without any execution optimization. In future research, we need to develop better algorithms for web services execution planning with respect to the following criteria: speed of execution, speed of connection, and cost to run.

Despite of these problems, the DALSystem has shown that it is possible to integrate computers seamlessly into the human problem solving process so that the domain users can perform their computation at the domain logical level without worrying about the underlying computer systems.

APPENDIX A
DALSYSTEM USER MANUAL

A.1 Introduction

The DALSystem is a brain assistant which allows computer users to interact with their concepts in domain ontologies. These concepts are linked with their implementation artifacts such as web services so that the users can perform their computation. A user can interact with her Ontology Manager via a program called DALConsole. This program can look up concepts in user ontology and help the user execute them.

After you have logged in to the system, you can use the tutorial in the following section to play with your concepts.

A.2 DALConsole Tutorial

A.2.1 Getting Started

The best way to learn a language is to write the Hello World.

In DALConsole, after the prompt, you type:

```
> print("Hello World!");
```

It will print

```
Hello World!
```

To list all the concepts in your ontology, type:

```
>list
```

The user can query information about her concept by the command

```
>info <concept name>
```

For example, if the user wants to know more about the concept `gcd`, she can type:

```
1 >info gcd
2 Concept Description: This is the function to find greatest common
      divisor (GCD)
3 of two integers. It is implemented using Euclidian algorithm.
4 Input information: NONE
5 Parameters Information: There are 2 parameter(s).
6 Parameter a of type http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
    Integer
7 Parameter b of type http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
    Integer
```

To call a concept, e.g. `gcd`, you write the concept name followed by the list of input parameters for that concept. For example, to call the concept `gcd` with two input parameters, 28 and 42, you type after the DALConsole prompt:

```
>gcd(28, 42)

14
```

You should get the answer of 14.

A.2.2 Variables and Arithmetic Expressions

Variables are declared using the following syntax

```
varName: type;
```

For example,

```
> x: integer;

> y: real;
```

In the domain of Arithmetic on Bula1, there are currently 4 types of primitive data concepts:

- integer
- real
- string
- boolean: for logical values (true, false)

After you declared a variable, you can use it in your computation. Arithmetic operators provided in DALConsole are:

$+, -, *, /$	arithmetic operators for integer, real values
and, or, not	logical operators for boolean
$==, =, <, >$	comparison operators for arithmetic expressions

Table A.1: DAL operators

So you can type after the prompt:

```
> 1 + 1

2

> 4.5 * 9.0

40.5

> 3.6 / 9.0

0.4
```

Notice that there is no semi colon after these expressions.

With variables, you can assign values to them by the operator “=”:

```
> z: integer;
> z = 16 * 23 - 4 / 2;
> print(z);
366
```

You can also print variable value just by typing the variable name without a semi colon at the end.

```
> z
366
```

A.2.3 Arrays

DALConsole currently doesn't allow you to create a new array type (you can create one using the addData2Onto command though). But you can use existing array type such as Vector (an array of 3 real numbers). To construct a vector variable with three elements 1.5, 2.4, 3.6, you type in:

```
> t: Vector;
> t = {1.5, 2.4, 3.6};
```

You can then print the content just like other variables by typing:

```
> print(t);
[1.5, 2.4, 3.6,]
```

Or simply

```
> t

[1.5, 2.4, 3.6,]
```

To access the i-th element of a vector (array), you use the [...] operator. For example, to get the second element of the vector t, you type

```
> t[2]

2.4
```

The system should print the value of the second element of t which is 2.4.

Now let's compute the length of vector t.

```
> sqrt(t[1] * t[1] + t[2] * t[2] + t[3] * t[3])

4.178516483155236
```

Now we can test the matrix with HouseHolderReduce algorithm. For example, if you want to solve the following linear equation system:

$$\begin{pmatrix} 2.0 & 2.0 & 4.0 \\ 1.0 & 3.0 & 1.0 \\ 3.0 & 1.0 & 3.0 \end{pmatrix} x = \begin{pmatrix} 18.0 \\ 1.0 \\ 14.0 \end{pmatrix}$$

$$Ax = b$$

In HouseHolderReduce algorithm, vectors are columns of the matrix. So in DALConsole, you type:

```
> a: Matrix;

> b: Vector;

> x: Vector;
```

```
> a[1] = {2.0, 1.0, 3.0};
> a[2] = {2.0, 3.0, 1.0};
> a[3] = {4.0, -2.0, 3.0};
> b = {18.0, 1.0, 14.0};
> x = HouseHolderReduce(a, b, 3);
```

A.3 DAL Language Tutorial

To go beyond the normal usage of DALConsole and create your own concept, you need to learn the DAL Language for Arithmetic Domain. There are two common types of concepts in one's computation domain: data concepts and action concepts. Data concepts such as integer, real, vector, etc. are concepts which hold data. Action concepts such as gcd, add, multiply, etc. manipulate data concepts and produce output results. The following sections will instruct you to create such concepts.

A.3.1 Creating action concepts

To create a action concept of your own, say addition of two vectors, there are two steps you need to do:

1. Write the description of your algorithm in a text file using an editor (such as `vi`), says `addV.dal`, then
2. From DALConsole prompt, you type the command,

```
>add2Onto addV.dal
```

For example, the content of the algorithm for adding two vectors can be written as follows:

```
1  concept: "addV";
2  description: "This is the concept of adding two vectors.";
3  message input: "Please enter two vectors.";
4  input: v1: Vector, v2: Vector;
5  output: v: Vector;
6  local: i: integer;
7
8  for i = 1; if i <= 3
9    begin
10     v[i] = v1[i] + v2[i];
11   end
12   withNext i = i + 1;
```

In this file, the first two lines are required to provide description about your concept. The third line is optional for displaying input information when a user requests information about this concept.

The next three lines (4 – 6) are optional for declaring the input, output and local variables that you may want to use in the algorithm. In each line, variable descriptions are separated by a comma and the whole line ends with a semicolon. So the line 4 means, your algorithm has two input parameters, v1, v2, of the type `Vector`.

After that is the main body of your algorithm. In this case there is a for-loop which compute the sum of two vectors v1 and v2. The details of this for-loop will be discussed later in section A.3.2.2.

You can then add this concept to your ontology using the DALConsole program with the command

```
>add2Onto addV.dal
```

After that you should use the `list` command to check if the concept `addV` is correctly added to your ontology. You can also query the information about this concept by typing:

```
>info addV
Concept Description: This is the concept of adding two vectors.
Input information: Please enter two vectors.
Parameters Information: There are 2 parameter(s).
Parameter v1 of type http://localhost:8080/cuongbk.owl#Vector
Parameter v2 of type http://localhost:8080/cuongbk.owl#Vector
```

Now let's use this concept to add two vectors x = {1.0, 2.0, 3.0} and y = {4.0, 5.0, 6.0}.

```
>x: Vector;
>x =  {1.0, 2.0, 3.0};
>y: Vector;
>y = {4.0, 5.0, 6.0};
>z: Vector;
>z = addV(x, y);
>print(z);
[5.0,7.0,9.0,]
```

A.3.2 Control-flow constructs

A.3.2.1 If-then

Formally, the syntax for **if-then** construct is

```
if expr then
   statement1(s);
else
  statement2(s);
endif;
```

Where `expr` is a boolean expression. If `expr` is evaluated to true, then `statement1(s)` are executed. Otherwise, `statement2(s)` are executed. The **else** branch is optional.

For example,

```
if n < 3 then
  z = a;
else
  z = b;
endif;
```

A.3.2.2 While and For

In the **while-loop** construct of

```
while expr do
  statement(s);
endwhile;
```

if the boolean expression `expr` is evaluated to `true`, the `statement(s)` is executed and the expression is re-evaluated. This cycle repeats until `expr` becomes `false`.

For example, the Euclidean algorithm for finding gcd of two integers a, b are expressed by **while-loop** as follows:

```
while b != 0 do
  t = b;
  b = a % b;
  a = t;
  print(a);
endwhile;
```

The **for-loop** has the formal syntax as:

```
for expr1; if expr2
  begin
    statement(s);
  end
  withNext expr3;
```

which is equivalent to

```
expr1;
while expr2 do
    statement(s);
    expr3;
```

```
endwhile;
```

For example, the addition of two vectors can be written as

```
for i = 1; if i <= 3
  begin
    v[i] = v1[i] + v2[i];
  end
  withNext i = i + 1;
```

A.3.3 Creating data concepts

To create a new data concept, say `Vector`, similarly to action concepts, there are also two steps you need to do:

1. Write the description of your data concept in a text file, say `vector.dal`, then
2. From DALConsole prompt, you type the command,

   ```
   >addData2Onto vector.dal
   ```

   Even though the two steps are very similar to those of action concepts, the key difference is the command `addData2Onto` instead of `add2Onto` (without keyword Data).

There are two types of composed data concepts: `arrays` and `records`.

A.3.3.1 Data concepts of Array type

The `Vector` concept is an array of real numbers. The formal syntax for declaring an array type is

```
concept <name> is
    array (lowerbound .. upperbound) of <base-type>;
endconcept;
```

where `<name>` is the concept name, `lowerbound, upperbound` are integer numbers specifying the range of indexes, `<base-type>` is the type of each element in the array.

For example the `Vector` concept in 3D space is defined as

```
concept Vector is
    array (1 .. 3) of real;
endconcept;
```

The whole vector.dal file looks like

```
1 concept: "Vector";
2 description: "Vector type";
3 local:
4 concept Vector is
5     array (1 .. 3) of real;
6 endconcept;
```

The first two lines are required to describe the concept.

A.3.3.2 Data concepts of Record type

The `Complex` concept is a record type with two fields `realPart` and `imgPart`. Let's examine the file `complex.dal` in detail.

```
1 concept: "Complex";
2 description: "Complex concept in complex analysis.";
3 local: concept Complex is
4   record
5      ImgPart : real;
6      RealPart : real;
7   endrecord;
8 endconcept;
```

The first two lines are required to describe the concept. The lines from 3 to 8 declare the concept `Complex` is of the record type with two fields ImgPart and RealPart, both are real numbers. `concept, is, record, endrecord, endconcept` are all keywords.

APPENDIX B
CEAD ONTOLOGY OWL FILE

Listing B.1: Action concept add definition in OWL

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:protege="http://protege.stanford.edu/plugins/owl/
        protege#"
    xmlns:xsp="http://www.owl-ontologies.com/2005/08/07/xsp.owl#"
    xmlns="http://bula1.cs.uiowa.edu/owl/cead.owl#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:p1="http://www.daml.org/services/owl-s/1.1/Profile.owl#
        "
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:swrl="http://www.w3.org/2003/11/swrl#"
    xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://bula1.cs.uiowa.edu/owl/cead.owl">

  <owl:Class rdf:ID="ActionConcept"/>
  <owl:Class rdf:ID="FilterConcept" >
      <rdfs:subClassOf rdf:resource="#ActionConcept" />
  </owl:Class>
  <owl:Class rdf:ID="DataConcept"/>
  <owl:Class rdf:ID="ComposedDataConcept">
      <rdfs:subClassOf rdf:resource="#DataConcept" />
  </owl:Class>
  <owl:Class rdf:ID="UnconstrainedArray">
      <rdfs:subClassOf rdf:resource="#DataConcept" />
  </owl:Class>
  <owl:Class rdf:ID="ConstrainedArray">
      <rdfs:subClassOf rdf:resource="#UnconstrainedArray" />
  </owl:Class>
  <owl:Class rdf:ID="PrimitiveDataConcept">
      <rdfs:subClassOf rdf:resource="#DataConcept" />
  </owl:Class>
  <owl:Class rdf:ID="Field"/>
  <owl:Class rdf:ID="Concept">
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#DataConcept" />
        <owl:Class rdf:about="#ActionConcept" />
      </owl:unionOf>
```

Listing B.1 continued

```
38    </owl:Class>
39    <owl:ObjectProperty rdf:ID="inputFilter">
40      <rdfs:domain rdf:resource="#DataConcept"/>
41      <rdfs:range rdf:resource="#ActionConcept"/>
42    </owl:ObjectProperty>
43    <owl:ObjectProperty rdf:ID="outputFilter">
44      <rdfs:domain rdf:resource="#DataConcept"/>
45      <rdfs:range rdf:resource="#ActionConcept"/>
46    </owl:ObjectProperty>
47    <owl:Class rdf:ID="Input"/>
48    <owl:Class rdf:ID="ServiceInstance" />
49    <owl:Class rdf:ID="Agent"/>
50    <owl:ObjectProperty rdf:ID="hasOutput">
51      <rdfs:domain rdf:resource="#ActionConcept"/>
52      <rdfs:range rdf:resource="#DataConcept"/>
53      <owl:inverseOf>
54        <owl:ObjectProperty rdf:ID="output"/>
55      </owl:inverseOf>
56    </owl:ObjectProperty>
57    <owl:ObjectProperty rdf:ID="hasInput">
58      <rdfs:domain rdf:resource="#ActionConcept"/>
59      <rdfs:range rdf:resource="#Input"/>
60    </owl:ObjectProperty>
61    <owl:ObjectProperty rdf:ID="inputType">
62      <rdfs:domain rdf:resource="#Input"/>
63      <rdfs:range rdf:resource="#DataConcept"/>
64    </owl:ObjectProperty>
65    <owl:ObjectProperty rdf:ID="implementedBy">
66      <rdfs:domain rdf:resource="#Agent"/>
67      <rdfs:range rdf:resource="#ServinceInstance"/>
68    </owl:ObjectProperty>
69    <owl:DatatypeProperty rdf:ID="inputName">
70      <rdfs:domain rdf:resource="#Input"/>
71      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
          string"/>
72    </owl:DatatypeProperty>
73    <owl:DatatypeProperty rdf:ID="order">
74      <rdfs:domain rdf:resource="#Input"/>
75      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
          int"/>
76    </owl:DatatypeProperty>
77    <owl:ObjectProperty rdf:ID="hasAgent">
78      <rdfs:domain rdf:resource="#ActionConcept"/>
79      <rdfs:range rdf:resource="#Agent"/>
```

Listing B.1 continued

```
80    </owl:ObjectProperty>
81    <owl:DatatypeProperty rdf:ID="description">
82      <rdfs:domain rdf:resource="#Concept"/>
83      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
            string"/>
84    </owl:DatatypeProperty>
85    <owl:DatatypeProperty rdf:ID="inputMessage">
86      <rdfs:domain rdf:resource="#ActionConcept"/>
87      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
            string"/>
88    </owl:DatatypeProperty>
89    <owl:DatatypeProperty rdf:ID="outputMessage">
90      <rdfs:domain rdf:resource="#ActionConcept"/>
91      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
            string"/>
92    </owl:DatatypeProperty>
93    <owl:DatatypeProperty rdf:ID="dataType">
94      <rdfs:domain rdf:resource="#DataConcept"/>
95      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
            string"/>
96    </owl:DatatypeProperty>
97    <owl:DatatypeProperty rdf:ID="hasName">
98      <rdfs:domain rdf:resource="#Field"/>
99      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
            string"/>
100   </owl:DatatypeProperty>
101   <owl:ObjectProperty rdf:ID="hasType">
102     <rdfs:domain rdf:resource="#Field"/>
103     <rdfs:range rdf:resource="#DataConcept"/>
104   </owl:ObjectProperty>
105   <owl:ObjectProperty rdf:ID="hasField">
106     <rdfs:domain rdf:resource="#ComposedDataConcept"/>
107     <rdfs:range rdf:resource="#Field"/>
108   </owl:ObjectProperty>
109   <owl:ObjectProperty rdf:ID="hasBaseType">
110     <rdfs:domain rdf:resource="#UnconstrainedArray"/>
111     <rdfs:range rdf:resource="#DataConcept"/>
112   </owl:ObjectProperty>
113   <owl:DatatypeProperty rdf:ID="hasLowerBound">
114     <rdfs:domain rdf:resource="#ConstrainedArray"/>
115     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
            int"/>
116   </owl:DatatypeProperty>
117   <owl:DatatypeProperty rdf:ID="hasUpperBound">
```

Listing B.1 continued

```
118     <rdfs:domain rdf:resource="#ConstrainedArray"/>
119     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
            int"/>
120   </owl:DatatypeProperty>
121   <owl:FunctionalProperty rdf:ID="serviceName">
122     <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
            DatatypeProperty"/>
123     <rdfs:domain rdf:resource="#ServiceInstance"/>
124     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
            string"/>
125   </owl:FunctionalProperty>
126   <owl:FunctionalProperty rdf:ID="wsdlFile">
127     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
            string"/>
128     <rdfs:domain rdf:resource="#ServiceInstance"/>
129     <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
            DatatypeProperty"/>
130   </owl:FunctionalProperty>
131   <owl:FunctionalProperty rdf:ID="uri">
132     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
            string"/>
133     <rdfs:domain rdf:resource="#ServiceInstance"/>
134     <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
            DatatypeProperty"/>
135   </owl:FunctionalProperty>
136   <owl:FunctionalProperty rdf:ID="operationName">
137     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
            string"/>
138     <rdfs:domain rdf:resource="#ServiceInstance"/>
139     <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
            DatatypeProperty"/>
140   </owl:FunctionalProperty>
141   <owl:FunctionalProperty rdf:ID="portName">
142     <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
            DatatypeProperty"/>
143     <rdfs:domain rdf:resource="#ServiceInstance"/>
144     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
            string"/>
145   </owl:FunctionalProperty>
146   <PrimitiveDataConcept rdf:ID="String">
147     <description>This is the primitive String concept for use
            with input filter.
148       </description>
```

Listing B.1 continued

```
149     <dataType rdf:datatype="http://www.w3.org/2001/XMLSchema#
            string">
150         xsd:string
151     </dataType>
152   </PrimitiveDataConcept>
153   <UnconstrainedArray rdf:ID="StringArray">
154     <description>This is the primitive String array concept for
            use with input filter.
155         </description>
156     <hasBaseType rdf:resource="#String" />
157   </UnconstrainedArray>
158 </rdf:RDF>
159
160 <!-- Created with Protege (with OWL Plugin 3.3.1, Build 430)
      http://protege.stanford.edu -->
```

APPENDIX C
HOUSEHOLDER REDUCTION ALGORITHMS

The following files are supposed to be added to user ontology in the order of appearance.

File product.dal:

```
concept: "product";
description: "compute the scalar product of two vectors from a
   position to the end.";
input: a: Vector, b: Vector, i: integer, n: integer;
output: c: real;
local: ab: real, k: integer;

ab = 0.0;
for k = i; if k <= n
  begin
    ab = ab + (a[k]) * (b[k]);
  end
  withNext k = k + 1;
c = ab;
```

Listing C.1: Compute the scalar product of two vectors

File eliminate.dal:

```
concept: "eliminate";
description: "Compute HouseHolder elimination.";
input: ai: Vector, vi: Vector, i: integer, n: integer;
output: c: Vector;
local: anorm: real, dii: real, fi: real, wii: real, k: integer;
anorm = sqrt(product(ai, ai, i, n));
if ai[i] > 0.0 then
  dii = -anorm;
else
  dii = anorm;
endif;
wii = ai[i] - dii;
fi = sqrt(-2.0 * wii * dii);
vi[i] = wii/fi;
ai[i] = dii;

for k = i + 1; if k <= n
  begin
    vi[k] = ai[k]/fi;
    ai[k] = 0.0;
  end
  withNext k = k + 1;

c = ai;
```

Listing C.2: HouseHolder elimination concept

File eliminate2.dal:

```
concept: "eliminate2";
description: "Compute HouseHolder elimination.";
input: ai: Vector, vi: Vector, i: integer, n: integer;
output: c: Vector;
local: anorm: real, dii: real, fi: real, wii: real, k: integer;
anorm = sqrt(product(ai, ai, i, n));
if (ai[i]) > 0.0 then
  dii = -anorm;
else
  dii = anorm;
endif;
wii = (ai[i]) - dii;
fi = sqrt(-2.0 * wii * dii);
vi[i] = wii/fi;
ai[i] = dii;

for k = i + 1; if k <= n
  begin
    vi[k] = ai[k]/fi;
    ai[k] = 0.0;
  end
  withNext k = k + 1;
c = vi;
```

Listing C.3: HouseHolder elimination concept

File transform.dal:

```
concept: "transform";
description: "HouseHolder transformation concept.";
input: aj: Vector, vi: Vector, i: integer, n: integer;
output: c: Vector;
local: fi:real, k: integer;

fi = 2.0 * product(vi, aj, i, n);
for k = i; if k <= n
  begin
    aj[k] = (aj[k]) - fi * (vi[k]);
  end
  withNext k = k + 1;
c = aj;
```

Listing C.4: HouseHolder transformation concept

File houseHolderReduce.dal:

```
concept: "HouseHolderReduce";
description: "HouseHolder Linear Equation System Solver concept."
   ;
input: a: Matrix, b: Vector, n: integer;
output: x: Vector;
local: vi: Vector, i: integer, j: integer, t1: Vector, t2: Vector
   , t: real, u: real;

for i = 1; if i <= n
  begin
    vi[i] = 0.0;
  end
  withNext i = i + 1;

for i = 1; if i < n
  begin
    t1 = eliminate(a[i], vi, i, n);
    t2 = eliminate2(a[i], vi, i, n);
    a[i] = t1;
    vi = t2;
    for j = i + 1; if j <= n
      begin
        a[j] = transform(a[j], vi, i, n);
      end
      withNext j = j + 1;
    b = transform(b, vi, i, n);
  end
  withNext i = i + 1;

for i = n; if 1 <= i
  begin
    t = 0.0;
    for j = i + 1; if j <=n
      begin
        t = t + x[j] * (a[j])[i];
      end
      withNext j = j + 1;
    x[i] = (b[i] - t) / a[i][i];
  end
  withNext i = i - 1;
```

Listing C.5: HouseHolder Linear Equation System Solver concept

File linearEquationSolver.dal:

```
concept: "LinearEquationSolver";
description: "This is implemented by HouseHolder reduction.";
output: t: Vector;
local: a: Matrix, b: Vector, x: Vector;

a[1] = {2.0, 1.0, 3.0};
a[2] = {2.0, 3.0, 1.0};
a[3] = {4.0, -2.0, 3.0};

b = {18.0, 1.0, 14.0};

x = HouseHolderReduce(a, b, 3);

t = x;
```

Listing C.6: A test for HouseHolder Linear Equation System Solver concept

APPENDIX D
SADL CODE FOR EUCLIDEAN ALGORITHM

Listing D.1: SADL code for Euclidean algorithm

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <sadl xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <declaration>
4      <imports>
5        <importOntology uri="http://localhost:8080/OntologyManager/
             ontologies/arithmeticCEAD.owl" />
6        <importOntology uri="http://localhost:8080/OntologyManager/
             ontologies/cead.owl" />
7        <importOntology uri="http://localhost:8080/OntologyManager/
             resources/sadl/cuongbk-CEAD.owl" />
8        <importOntology uri="http://bula1.cs.uiowa.edu/owl/
             arithemticCEAD.owl" local="file:../../owl/arithmeticCEAD
             .owl" />
9      </imports>
10     <inputs>
11       <input type="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
             Integer" index="2" />
12       <input type="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
             Integer" index="3" />
13     </inputs>
14     <outputs>
15       <output type="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
             Integer" index="4" />
16     </outputs>
17   </declaration>
18   <init type="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
       Integer" index="1" />
19   <init type="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
       Integer" index="4" />
20   <load index="2" />
21   <printTOS />
22   <load index="3" />
23   <printTOS />
24   <label name="label2" />
25   <load index="3" />
26   <pushStr value="0" />
27   <loadConst conceptURI="http://bula1.cs.uiowa.edu/owl/arithmetic
       .owl#Integer" />
```

Listing D.1 continued

```
28   <notEqualI xmlns="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#
        " params="2" />
29   <jumpfalse label="label1" />
30   <load index="3" />
31   <store index="1" />
32   <load index="2" />
33   <load index="3" />
34   <modI xmlns="http://bula1.cs.uiowa.edu/owl/arithmetic.owl#"
        params="2" />
35   <store index="3" />
36   <load index="1" />
37   <store index="2" />
38   <load index="2" />
39   <printTOS />
40   <jump label="label2" />
41   <label name="label1" />
42   <load index="2" />
43   <store index="4" />
44 </sadl>
```

REFERENCES

Agarwal, V., Dasgupta, K., Karnik, N., Kumar, A., Kundu, A., Mittal, S. & Srivastava, B. (2005), A service creation environment based on end to end composition of web services, *in* 'Proceedings of the 14th international conference on World Wide Web', WWW '05, ACM, New York, NY, USA, pp. 128–137.
URL: http://doi.acm.org/10.1145/1060745.1060768

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

Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B. & Mock, S. (2004), Kepler: An extensible system for design and execution of scientific workflows, *in* 'IN SSDBM', pp. 21–23.

Andrews, T. & et al. (2003), 'Business process execution language for web services'.
URL: https://www.oasis-open.org/committees/download.php/2046/BPEL%20V1-1%20May%205%202003%20Final.pdf

Baader, F., Horrocks, I. & Sattler, U. (2007), Chapter 3: Description Logics, *in* F. van Harmelen, V. Lifschitz & B. Porter, eds, 'Handbook of Knowledge Representation', Elsevier.

Backus, J. W. (1959), The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference, *in* 'IFIP Congress', pp. 125–131.

Biermann, A. W. & Ballard, B. W. (1980), 'Toward natural language computation', *Comput. Linguist.* **6**, 71–86.
URL: http://portal.acm.org/citation.cfm?id=972439.972440

Bischof, M., Kopp, O., van Lessen, T. & Leymann, F. (2009), BPELscript: A Simplified Script Syntax for WS-BPEL 2.0, *in* '2009 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009)', IEEE Computer Society Press, pp. 39–46.
URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2009-49&engl=0

Boisvert, A., Arkin, A. & Riou, M. (2008), 'Bpel simplified syntax'.
URL: https://cwiki.apache.org/ODExSITE/bpel-simplified-syntax-simbpel.html

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte,

S. & Winer, D. (2000), ‘Simple Object Access Protocol (SOAP) 1.1’.
URL: http://www.w3.org/TR/soap/

Christensen, E., Curbera, F., Meredith, G. & Weerawarana, S. (2001), ‘Web Services Description Language (WSDL) 1.1’.
URL: http://www.w3.org/TR/wsdl.html

Cickovski, T. M. (2004), Biologo, a domain-specific language for morphogenesis, Master’s thesis, Computer Science and Engineering, University of Notre Dame.

Clement, L., Hately, A., von Riegen, C. & Rogers, T. (2004), ‘Uddi version 3.0.2 spec’.
URL: http://uddi.org/pubs/uddi_v3.htm

Craig, I. D. (2006), *Virtual machines*, Springer-Verlag.

Curcin, V. & Ghanem, M. (2008), Scientific workflow systems - can one size fit all?, *in* ‘Biomedical Engineering Conference, 2008. CIBEC 2008. Cairo International’, pp. 1–9.

Curtis, D. E., Rus, T. & Jensen, J. (2008), Application driven software for chemistry, *in* ‘2008 IEEE International Conference on Electro/Information Technology, EIT 2008, held at Iowa State University, Ames, Iowa, USA, May 18-20, 2008’, IEEE, pp. 361–366.

Deelman, E., Singh, G., hui Su, M., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C. & Katz, D. S. (2005), ‘Pegasus: a framework for mapping complex scientific workflows onto distributed systems’, *SCIENTIFIC PROGRAMMING JOURNAL* **13**, 219–237.

Deursen, A. V., Klint, P. & Visser, J. (2000), ‘Domain-specific languages: An annotated bibliography’, *ACM SIGPLAN NOTICES* **35**, 26–36.

DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J. & Hanrahan, P. (2011), Liszt: a domain specific language for building portable mesh-based pde solvers, *in* ‘Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis’, SC ’11, ACM, New York, NY, USA, pp. 9:1–9:12.
URL: http://doi.acm.org/10.1145/2063384.2063396

DuCharme, B. (2011), *Learning SPARQL*, O’Reilly Media.

Erl, T., Balasubramanian, R., Carlyle, B. & Pautasso, C. (2012), *SOA with REST: Princi-*

ples, Patterns and Constraints for Building Enterprise Solutions with REST, Prentice Hall.

Fischer, C. N., Cytron, R. K. & LeBlanc, R. J. (2010), *Crafting a Compiler*, Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gasevic, D., Djuric, D. & Devedzic, V. (2009), *Model Driven Engineering and Ontology Development*, 2nd edn, Springer Publishing Company, Incorporated.

Green, T. & Petre, M. (1992), When visual programs are harder to read than textual programs, *in* 'Sixth European Conference on Cognitive Ergonomics', pp. 167–180.

Gubala, T., Bubak, M., Malawski, M. & Rycerz, K. (2006), Semantic-based grid workflow composition, *in* 'Proceedings of the 6th international conference on Parallel Processing and Applied Mathematics', PPAM'05, Springer-Verlag, Berlin, Heidelberg, pp. 651–658.
URL: `http://dx.doi.org/10.1007/11752578_78`

Horn, P. (2001), AUTONOMIC COMPUTING: IBM's Perspective on the State of Information, Technical report, IBM Corporation.

Kiper, J. D., Auernheimer, B. & Ames, C. K. (1997), 'Visual depiction of decision statements: What is best forprogrammers and non-programmers?', *Empirical Softw. Engg.* **2**(4), 361–379.
URL: `http://dx.doi.org/10.1023/A:1009797801907`

Knöll, R. & Mezini, M. (2006), Pegasus: first steps toward a naturalistic programming language, *in* 'Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications', OOPSLA '06, ACM, New York, NY, USA, pp. 542–559.
URL: `http://doi.acm.org/10.1145/1176617.1176628`

Knuth, D. E. (1965), 'On the translation of languages from left to right', *Information and Control* **8**(6), 607 – 639.

Krishnan, S. & Bhatia, K. (2007), Soas for scientific applications: Experiences and challenges, *in* 'e-Science and Grid Computing, IEEE International Conference on', pp. 160–169.

Liu, H. & Lieberman, H. (2005*a*), Metafor: visualizing stories as code, *in* 'Proceedings of the 10th international conference on Intelligent user interfaces', IUI '05, ACM, New York, NY, USA, pp. 305–307.
URL: http://doi.acm.org/10.1145/1040830.1040908

Liu, H. & Lieberman, H. (2005*b*), Programmatic semantics for natural language interfaces, *in* 'CHI '05 extended abstracts on Human factors in computing systems', CHI '05, ACM, New York, NY, USA, pp. 1597–1600.
URL: http://doi.acm.org/10.1145/1056808.1056975

Lopes, C. V., Dourish, P., Lorenz, D. H. & Lieberherr, K. (2003), 'Beyond AOP: Toward Naturalistic Programming', *ACM SIGPLAN Notices* **38**(12), 34–43. OOPSLA'03 Special Track on Onward! Seeking New Paradigms & New Thinking.
URL: http://www.ccs.neu.edu/home/lorenz/papers/oopsla03a/

Margolis, B. (2007), *SOA for the Business Developer: Concepts, BPEL, and SCA*, Mc Press.

Martin, D. & et al (2003), 'Daml-s (and owl-s) 0.9 draft release'.
URL: http://www.daml.org/services/daml-s/0.9/

McCarthy, J. (1980), 'Circumscription - a form of non-monotonic reasoning', *Artificial Intelligence* **13**(1-2), 27 – 39. Special Issue on Non-Monotonic Logic.
URL: http://www.sciencedirect.com/science/article/pii/0004370280900119

McDermott, D. V. (2002), Estimated-regression planning for interactions with web services, *in* M. Ghallab, J. Hertzberg & P. Traverso, eds, 'AIPS', AAAI, pp. 204–211.

McGuinness, D. L. & van Harmelen, F. (2004), 'OWL Web Ontology Language'.
URL: http://www.w3.org/TR/owl-features/

McIlraith, S. A. & Son, T. C. (2002), Adapting golog for composition of semantic web services, *in* D. Fensel, F. Giunchiglia, D. L. McGuinness & M.-A. Williams, eds, 'KR', Morgan Kaufmann, pp. 482–496.

Mernik, M., Heering, J. & Sloane, A. M. (2005), 'When and how to develop domain-specific languages', *ACM Comput. Surv.* **37**(4), 316–344.
URL: http://doi.acm.org/10.1145/1118890.1118892

Miller, L. A. (1981), 'Natural language programming: Styles, strategies, and contrasts', *IBM Systems Journal* **211**(2), 184–215.

Mordani, R. (2009), Jsr 315: Javatm servlet 3.0 specification, Technical report, Oracle Corporation.
URL: http://www.jcp.org/en/jsr/detail?id=315

Myers, B. A., Pane, J. F. & Ko, A. (2004), 'Natural programming languages and environments', *Commun. ACM* **47**, 47–52.
URL: http://doi.acm.org/10.1145/1015864.1015888

Narayanan, S. & McIlraith, S. A. (2002), Simulation, verification and automated composition of web services, *in* 'Proceedings of the 11th international conference on World Wide Web', WWW '02, ACM, New York, NY, USA, pp. 77–88.
URL: http://doi.acm.org/10.1145/511446.511457

Oinn, T., Addis, M., Ferris, J., Marvin, D., Carver, T., Pocock, M. R. & Wipat, A. (2004), 'Taverna: A tool for the composition and enactment of bioinformatics workflows', *Bioinformatics* **20**, 2004.

Petre, M. (1995), 'Why looking isn't always seeing: readership skills and graphical programming', *Commun. ACM* **38**(6), 33–44.
URL: http://doi.acm.org/10.1145/203241.203251

Polya, G. (1945), *How to Solve It*, Princeton University Press.

Popek, G. J. & Goldberg, R. P. (1974), 'Formal requirements for virtualizable third generation architectures', *Commun. ACM* **17**(7), 412–421.
URL: http://doi.acm.org/10.1145/361011.361073

Price, D., Rilofff, E., Zachary, J. & Harvey, B. (2000), NaturalJava: a natural language interface for programming in Java, *in* 'Proceedings of the 5th international conference on Intelligent user interfaces', IUI '00, ACM, New York, NY, USA, pp. 207–211.
URL: http://doi.acm.org/10.1145/325737.325845

Qin, J. & Fahringer, T. (2008), A novel domain oriented approach for scientific grid workflow composition, *in* 'Proceedings of the 2008 ACM/IEEE conference on Supercomputing', SC '08, IEEE Press, Piscataway, NJ, USA, pp. 21:1–21:12.
URL: http://dl.acm.org/citation.cfm?id=1413370.1413392

Rao, J. & Su, X. (2004), A survey of automated web service composition methods, *in* 'In Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC 2004', pp. 43–54.

Ross, D. T. (1978), 'Origins of the apt language for automatically programmed tools', *SIGPLAN Not.* **13**(8), 61–99.
URL: http://doi.acm.org/10.1145/960118.808374

Rus, T. (1993), *System Software and Software Systems: Systems Methodology for System Software*, Vol. 1, World Scientific Pub Co Inc.

Rus, T. (2008), Liberate computer user from programming, *in* J. Meseguer & G. Rosu, eds, 'Algebraic Methodology and Software Technology, 12th International Conference, AMAST 2008, Urbana, IL, USA, July 28-31, 2008, Proceedings', Vol. 5140 of *Lecture Notes in Computer Science*, Springer, pp. 16–35.

Rus, T. (2013), Computer integration within problem solving process, *in* 'Proceedings of RoEduNet 11th International Conference: Networking in Education and Research', Sinaia, Romania.

Rus, T. & Bui, C. (2010), Software development for non-expert computer users, *in* 'Proceedings of the International Conference on Cloud Computing and Virtualization', CCCV '2010, Singapore.

Rus, T. & Curtis, D. E. (2006), Application driven software development, *in* 'Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006), October 28 - November 2, 2006, Papeete, Tahiti, French Polynesia', IEEE Computer Society.

Rus, T. & Curtis, D. E. (2007), Toward application driven software technology, *in* H. R. Arabnia & H. Reza, eds, 'Proceedings of the 2007 International Conference on Software Engineering Research & Practice, SERP 2007, Volume I, June 25-28, 2007, Las Vegas Nevada, USA', CSREA Press, pp. 282–288.

Sammet, J. E. (1966), 'The use of english as a programming language', *Commun. ACM* **9**, 228–230.
URL: `http://doi.acm.org/10.1145/365230.365274`

Scanlan, D. A. (1989), 'Structured flowcharts outperform pseudocode: An experimental comparison', *IEEE Softw.* **6**(5), 28–36.
URL: `http://dx.doi.org/10.1109/52.35587`

Sipser, M. (2006), *Introduction to Theory of Computation*, Cengage.

Srivastava, B. & Koehler, J. (2003), Web service composition - current solutions and open problems, *in* 'In: ICAPS 2003 Workshop on Planning for Web Services', pp. 28–35.

Taylor, I. J., Wang, I., Shields, M. S. & Majithia, S. (2005), 'Distributed computing with triana on the grid', *Concurrency - Practice and Experience* **17**(9), 1197–1214.

Welty, C. & Guarino, N. (2001), 'Supporting ontological analysis of taxonomic relation-

ships', *Data and Knowledge Engineering* **39**(1), 51 – 74. 19th International Conference on Conceptual Modeling (ER2000).

Wolfram, S. (2010), 'Programming with natural language is actually going to work'.
URL: http://blog.wolfram.com/2010/11/16/programming-with-natural-language-is-actually-going-to-work/

Woods, W. A. (1970), 'Transition network grammars for natural language analysis', *Commun. ACM* **13**, 591–606.
URL: http://doi.acm.org/10.1145/355598.362773