Application Driven Software Development

Teodor Rus and Donald Ephraim Curtis
Department of Computer Science
University of Iowa, Iowa City, IA 52242

I. Rationale for Application Driven Software Development

Even in its very infancy computer technology has been seen as a collection of tools destined to solve problems of a given application domain (AD) [1]. The problem solving process using computers is (and has been) carried out within the computer environment and requires the AD experts to formalize their problems in computer terms. The effort put forth so far toward making this process easier for AD experts has generated a rich and well-defined information technology (IT) domain, populated by computer artifacts such as programming languages and program generation tools. Successes of this approach to problem-solving led to the development of current computer technology whose complexity overwhelms computer experts themselves. Nevertheless, the usage of current IT for problem solving still requires AD experts to manipulate IT domain concepts and tools rather than AD concepts and tools. To further help this process, more and more complex IT tools are generated thus increasing software complexity to a level where only with formidable difficulties can AD experts manage to develop their application systems. Among the side effects of this situation are the lack of efficiency in application system development, poor performance in computer utilization, and even threat to the future evolution of computer technology itself. Our conjecture is that in order to break this vicious circle we need to rethink the problem solving process. We need to abandon the requirement that AD experts manipulate computer terms and to allow them to manipulate AD specific terms using AD specific languages. The recent advances created by computing research makes it feasible to move the problem solving process from the IT domain into the AD domain.

To understand the difference between our research toward *application driven software development* and previous similar research we are looking first at three research directions that have the same goal: problem solving methods (PSM) [FEMR96], [FM01], [FMvH+03], [CM04] carried out in the artificial intelligence community, model driven architectures (MDA) [SK97], [Fra], [GBI+04], [Coo04], [BGK+06] carried out in the software engineering community, and domain-specific languages (DSL) [SLCG99], [GM03], [CCMWa], [vDKV] carried out mostly in the academic community.

Problem solving methods provide reusable architectures and components for implementing the reasoning part of knowledge-based systems. Since these architectures and components manipulate computer artifacts the term “problem-solving” as used in PSM is a synonym to the term ”program-development” used in IT. That is, the goal of PSM is to develop automated methods to generate programs that solve given problems. Model driven architectures [SK97] are currently used in software engineering as a generic term for describing the use of models within the software engineering process. The foundation of MDA [GBI+04] is based on three ideas: (1) direct representation of the ideas and concepts of problem domain, (2) use computer-based tools to bridge the semantic gap between domain concepts and implementation technology, and (3) use open standards to eliminate useless diversity and encourage the production of general purpose tools. Models are computerized representations [Coo04] whose elements *correspond* to elements or concepts in the problem domain. However, modeling technology developed over the past years, such as Unified Modeling Language (UML), uses as representation elements *IT representations* not *AD concepts*. Domain-specific languages are designed so that they can more directly represent the problem domain which is being addressed. That is, domain-specific languages help IT experts to handle application domain concepts rather than helping AD experts handle computer technology. The development of older programming languages (Cobol, Fortran, Lisp) all came onto existence as dedicated languages for solving problems in certain areas of interest [vDKV]. The need for specialized language support to solve problems in well-defined application domains has resurfaced over and over again.

In summary, PSM, MDA, and DSL have been developed with the goal of helping the problem solving process. But since they operate with computer abstractions not with AD abstractions they cannot fully bridge the semantic gap between IT and AD. Therefore, we can safely conclude that the research on computer artifact development performed so far has as the goal to make computer artifact development more easy, more efficient, more everything. But note, computer artifact development is not really the objective of problem solving within a given problem domain.

Another important advance in software development is the realization that software complexity can be better handled by developing and studying software architectures that separate software descriptions from software implementation. Barry Boehm [Boe96] in his foreword for the book [SG96] identifies precisely the relationship between software architecture and the application oriented problem solving process when he observes that ”the biggest problem in software engineering is the shortage of intermediate abstractions that connect the

[1] An application domain is any field of human endeavor where computers may be used to solve problems.

characteristics of the systems users (AD experts) need to the characteristics of systems that software engineers can build". But software architectures could do more than provide the framework to separate software description from software implementation. Software architectures can also provide the framework for an application driven software development. Software architecture allows us to think about software systems as problem solving artifacts independently of their implementations, exactly as a car-building engineer thinks about a car independent of the differential equations describing its components. So far software architecture allows us to see the shortage of abstractions that connect AD to the IT domain, but it does not tell us precisely how we are to bridge this gap. However, looking at the problem solving process from the perspective of the architectural aspect of software systems that solve AD problems, one can see that this gap can be bridged by *computational emancipation of the application domain*.

By computational emancipation of an AD we mean two things: first, it means that AD is provided with a domain oriented computational structure and second, it means that the computational structure of the application domain is provided with IT semantics. Computational emancipation of AD can be achieved by structuring the AD using ontologies where concepts are associated with computer artifacts as semantics. AD ontology provides the abstractions AD experts need to express naturally their problems and solution algorithms while the computational semantics associated with AD concepts allow IT experts to develop software that map AD systems into computer artifacts that implement them. Thus, our research initiates a software development methodology where:

- AD expert handles AD terms (not UML or any other IT abstractions) to create AD-systems that provide solutions to AD applications.
- IT expert develops IT-systems that map automatically AD-systems into equivalent IT-systems performing them.

The framework for the design and implementation of such application driven software consists of:

1) Create ontology development tools and description logic languages. AD experts use use these tools to develop the AD ontology and AD and IT experts use description logic languages to associate ontological terms with computer artifacts as meaning.
2) Develop domain driven software architecture description languages (SADL) that use concepts of the AD ontology and provide them with interpreters that map SADL expressions into software systems that implement the systems described by such expressions.
3) Use this process extensively with various application domains. This can be done by hands-on computer technology approach of teaching where AD expert creates AD ontologies and AD software tools rather than creating prose. This is similar to what mathematicians do when they teach their field of interest by developing theorem proofing methodology rather than developing prose [2].

We use Acme [GMW00] as a model for SADL development. However, the scope of this research is much larger than what we can fit into a paper. Therefore, the goal of this paper is rather to illustrate our approach for application driven software development using an appropriate application domain. Biased by our own application domain we have chosen language processing as the AD. Hence, in section 2 we provide an open-ended ontology of language processing. Section 3 is dedicated to language processors description using a SADL and its interpreter. Section 4 presents a case-study that shows how can this approach take advantage of the current software technology, particularly provided by Web Service Description Languages (WSDL), for its implementation.

II. Computational Emancipation of Problem Domain

As stated in the introduction, computational emancipation of an application domain consists of providing it with a computational structure whose concepts are associated with software artifacts as their semantics. This abstraction allows us to move the process of problem solving with computers from the IT domain into the AD domain where it belongs. Here we are concerned with problem solving in the domain of language processing.

A. Computational emancipation of language processing

The first step in the process of computational emancipation of a language is the design of a language ontology. This can be represented using an appropriate T-box of a Description Logic [BCM+05] or, as we proceed here for simplicity, using an appropriate tree, Figure 1. The leaves of this tree represent the vocabulary of the ontology, interior nodes represent concepts constructed in terms of other concepts, and the arrows and the dotted lines represent relationships between the concepts in the ontology.

The elements of the vocabulary of the language ontology in Figure 1 are:

- The *Alphabet*, which is a finite set of distinguishable symbols.
- Lexical entities, which are strings of symbols over the Alphabet, usually split in classes that may not be disjoint. In natural languages these classes are tag-sets, such as nouns, verbs, adjectives, adverbs, etc. In programming languages they are lexical tokens, such as identifiers, numbers, keywords, operators, relations, separators, delimiters, parentheses, etc. Each such class is specified by either a dictionary (in natural languages) or by a regular expression over (a portion of) the alphabet in programming languages.
- Discourse is the collection of well-formed language-constructs classified as syntax categories by the specification rules. In natural languages these are phrases

[2] Note, in order to provide a systematic methodology for using computer technology to solve their problems even mathematicians need to computationally emancipate their working domains.

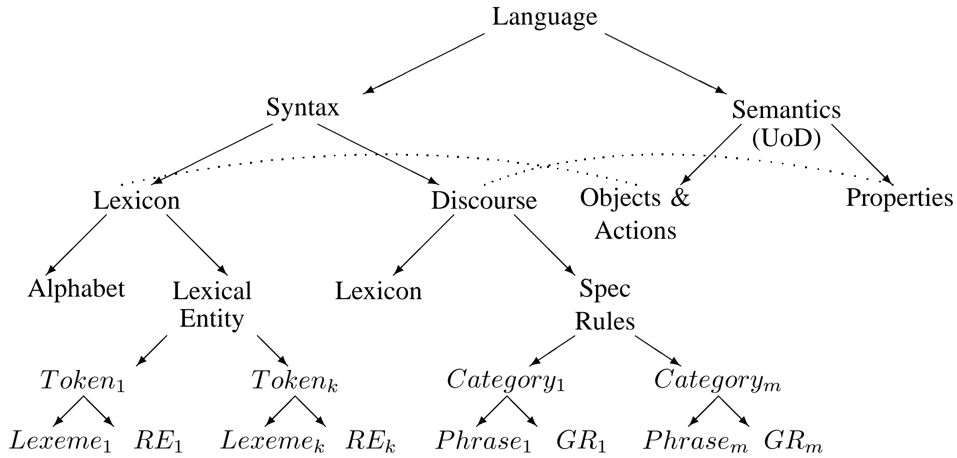

Fig. 1. Language ontology

constructed according to given grammatical rules. In programming languages these are constructs representing units of computations such as expressions, statements of various forms, functions, programs, etc. In both, natural and programming languages, the specification rules are most often provided by context-free grammars whose terminals are the lexical tokens (tag-sets).

- Objects and actions are elements of the universe of discourse (UoD) denoted by the lexical entities of the language. In programming languages these are domain of values handled by language constructs.
- Properties represent knowledge about the UoD and are expressed by language phrases. In programming languages these are language constructs representing computations.
- Syntax and Semantics are related by two operators $Value : Syntax \rightarrow Semantics$ and $Discourse : Semantics \rightarrow Syntax$ ($Eval$ and $Learn$ in [Rus02]) defined by equations of the form:

$$Value(Syntax.element) =$$
$$\{UoD.elements\ denoted\ by\ Syntax.element\}$$
$$Discourse(UoD.element) =$$
$$\{Syntax.elements\ denoting\ the\ UoD.element\}$$

Notice that an element of the universe of discourse may have multiple syntactic representations and a syntactic element may represent multiple elements of the universe of discourse.

The second step in the process of language emancipation consists of providing the nodes of the ontology with computational meanings. This is strongly related to the problem-solving process in the language domain.

We assume here that the computational meaning of the concepts used by the domain ontology are universal algorithms that solve classes of problems, are characteristic to the domain, and are mathematically proven correct. These assumptions are transformed by the problem solving process into the following characterization of the algorithms used as computational meaning of the AD concepts:

1) Computational meaning associated with an ontology node are stand-alone computation processes further referred to as *components*. Note, data are computation processes that leave their input unchanged.
2) The behavior of a computational process is completely defined by its input/output performance.
3) The interaction between computational processes is achieved by an appropriate combination of one or more of the mechanisms: calling patterns, sharing appropriate data or procedures, messaging systems.
4) Composition of computational processes is performed by filters that map the output generated by a component into the input expected by another component.

In the particular example of language processing domain, the nodes of the ontology tree in Figure 1 are associated with stand-alone computational entities such as, integer arithmetic, string manipulation operations, table-search algorithms, regular expressions generating strings and finite automata recognizing these strings, context free grammars generating phrases and push-down automata recognizing these phrases. This list is open-ended, and evolves with the problem solving process.

B. Example of a language processing problem and solution algorithm

A language user whose ontology is given may formulate the following problem.

> Create a tool that input a text and produces a stream of tokens replacing each lexical element of the text with its token. If a string in the input text is not recognized as a lexeme of the language the tool should print a token error such as NoT.

For example, the English text ”This is a test, please disregard” would be mapped into ”A V A N, V V”; the English text ”This

is a blabla, please disregard" would be mapped into "A V NoT N, V V"; the C language text "if x > y x = x-1 else y = y - 1" would be mapped into "if id > id id = id - nr else id = id - nr"

What knowledge would a language user need to have in order to develop such a tool and how could her solution be automatically mapped into a program that performs this computation on a computer?

Language ontology tells language user that every lexical element of a language text is specified by a regular expression and the token of that lexeme is a name given to that expression. The language user may also know that each regular expression is equivalent with a finite automaton and that a component that simulates that automaton is associated with the ontology tree node representing that regular expression. In other words, the language user has the black boxes $C_1, ...C_k$, associated with the language ontology nodes representing the language lexemes. In addition, the language user knows that each black box C_i inputs strings s_i over the language alphabet and outputs either the token t_i, if s_i is specified by the regular expression associated with C_i, or the special token NoT. Note that NoT is different from all other tokens. With this knowledge the language user may provide the following description of the proposed tool, called here the `Scanner`:

```
Scanner:
    Input: stream of lexemes;
Output: stream of tokens;
    Lexeme: string;
Token: token representation;
    Output = empty;
    while (more lexemes in the Input) repeat
        Lexeme = NextLexeme(Input);
        Token  = C_i(Lexeme) or NoT;
        Output = Output . Token;
```

Fig. 2. The scanner generation

where · denotes concatenation operator. Using this scanner the language processing expert can employ a computer to map a text into the corresponding string of tokens.

III. Application Driven Problem Solving Methodology

Traditional problem solving methodology requires the AD expert to produce programs written in a programming language implemented on a computer in order to employ that computer to solve a problem. The programming language is a high-level notation of the computations performed by the computer. To perform this task, the AD expert, in addition to expertise on her problem domain, needs to master three things: (1) understand machine computation, (2) express machine computations using the concepts provided by the programming language, and (3) encode the concepts of her problem domain as valid programs of the programming language. This is a lengthy and difficult process.

A. Application driven problems and solution algorithms

The goal of computational emancipation of an AD is to allow AD experts to use computer technology as a problem solving tool dedicated to their AD without requiring the AD expert to develop IT representations. Assuming that the computational emancipation of the ADs has been performed, the AD concepts are already associated with computations performed by the problem solving tool, the computer. This is similar to the manipulation of arithmetic expressions by mathematicians doing arithmetic. Arithmetic expressions need not be represented using other computational terms because they represent computations performed by the human brain while doing arithmetic. In other words, the tool (i.e., the computer) is seen here as an extension of the brain performing the processes represented by the AD concepts and thus helping the AD expert to manipulate her problem domain more efficiently.

Consequently the problem solver in a computationally emancipated AD manipulates AD terms which represent computation processes they understand and which are performed by the tool (the computer) exactly as arithmetic computations are performed by the brain. Better yet, AD computations are expressed using the natural language of the AD domain and are performed by the AD expert's brain using the computer as a powerful tool. Therefore, the language used by the AD expert to represent problems and solution algorithms should be the natural language of the AD expert. The software technology that IT domain experts need to develop, in order to make this approach to problem solving feasible, consists of interpreters dedicated to the problem domain which interpret the problem solutions, provided by AD experts, and map them into computer processes implementing these solutions. This is carried out using the computation processes associated with the terms used by AD experts in the AD ontology as components. This interpretation is based on a few new ideas which can be summarized as follows:

1) The solutions developed by AD experts contain two types of terms: (a) terms present in the AD ontology (such as regular expression, finite automaton, lexeme, token, etc., used in the example in Figure 2) and (b) terms that are not present in the AD ontology.
2) Terms present in the AD ontology are directly interpretable as computation processes performed by the computer. Terms that are not present in the ontology are treated by the interpreter as operators of process compositions. Computation performing these process compositions is specific to the interpreter itself and thus it is implemented by the interpreter.
3) The result of the computation performed by the interpreter composing processes implicitly or explicitly present in the expert solution can be one of:
    a) the expression of a persistent process that is associated with a new ontology term (such as *scanner*) thus performing *ontology expansion*;
    b) the transient process that implements the expert solution thus solving the problem (such as the

scanning process performed by the expression in Figure 2);

c) the value produced by the process generated by the interpreter, such as the stream of tokens produced by the computation expressed in Figure 2.

Note, in cases (b) and (c) above it is irrelevant which computer performs the processes initiated by the interpreter or what kind of language these computational processes are represented in. Thus, the usual problems faced by software development disappear. However, in case (a) where the interpreter generates code, the interpreter does need to face these problems. Therefore, in this situation the interpreter relies on language translation methodology.

To illustrate this discussion further, we consider here the problem of creating a language interpreter which inputs an (arithmetic) expression and maps it into the arithmetic value it represents. This interpreter uses as components a scanner and a parser. The solution to this problem requires the language user to handle the computation terms within the language processing domain which are not yet in the language processing ontology. Therefore here we assume that through previous developments the language ontology has been augmented with the language processing ontology given in Figures 3 and 4. The solution formulated by the language expert is in Figure 5.

```
Evaluator:
  Input: arithmetic expression of type text;
  Output: arithmetic value of type number;
  Intermediate Form:
          IF1 of type streamOf(Token,Lexeme);
  Intermediate Form:
          IF2 of type abstract syntax tree;
  Run:
    IF1 = scanner(Input);
    IF2 = parser(IF1);
    Output = evaluate(IF2);
```

Fig. 5. Expression evaluation

To understand the structure of the interpreter that maps this language processing solution algorithm into the value of its input, we must observe that the interpreter consists of two transformations:

1) A translator, that maps the solution algorithm supplied by the AD expert into an interpretable form written into an IT dependent language called here the Software Architecture Description Language (SADL).
2) The interpreter itself that maps SADL-expressions into the computations they represent.

The description of the translator requires both a formal specification of the language used by the domain expert to write its solution algorithms and a formal specification of the SADL. Since AD expert uses her natural language to express problems and solution algorithms, in this paper we focus on SADL and its mapping into the computations the SADL expressions represent.

B. *Software Architecture Description Language*

The SADL is the intermediate language we use to represent AD solutions in the IT domain. This language is organized on three levels of structuring:

1) The lexical elements of the language are the AD concepts and AD operators. The semantics of AD concepts used in SADL are the IT computation artifacts associated with them in the AD ontology. The semantics of AD operators are computations specified by SADL interpreter.
2) The second level of SADL is the computation process which is either specified by the signature of the computation artifacts used in the AD ontology or are operators that compose such processes in SADL.
3) The third level of SADL is the system which consists of sequential and parallel process compositions of one or more SADL processes that implement an AD solution algorithm.

The intermediary nature of SADL allows the AD problem solving process to evolve with the AD while SADL implementations evolve with advances provided by IT. It should also be noted that SADL is not meant for either the AD expert or the IT expert. SADL is simply provided as a way to bridge the AD solutions to IT processes implementing those solutions.

SADL design is inspired from both software architecture description languages, such as Acme [GMW00] and semantic web technologies, such as the web service description language (WSDL) [CCMWb] [CCMWa]. SADL borrows its semantics from Acme. However, we restrict the fundamental concepts SADL handles to three: *component*, which has the same meaning as component in Acme, *filter*, which has the flavor of a connector in Acme without concerns about ports and roles, and *system* which has the same meaning as system in Acme. SADL builds further on Acme by adding operators that allows SADL to express the control flow among the components of a SADL system. SADL borrows its syntax from XML rather than using C-like syntax as Acme does. The XML syntax used by SADL is specified as follows:

1) SADL lexical elements are the concepts used in the natural language of the AD expert.
2) A SADL process is represented by an XML element. The tags of XML elements representing SADL processes identify the nature of the computation performed. These tags are associated with an open-ended list of attributes (such as name, signature, location, etc.) which specify the process location and describe the environment in which the computations performed by the process operate.
3) A SADL system is then a sequence of XML elements that describe a composed process which represents one of: (a) the computer code that implement a new concept of the AD ontology, (b) the computer process that implement a user solution, (c) the value produced by a computer process.

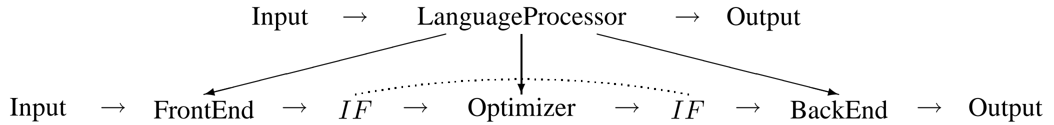

Fig. 3. A language processing ontology

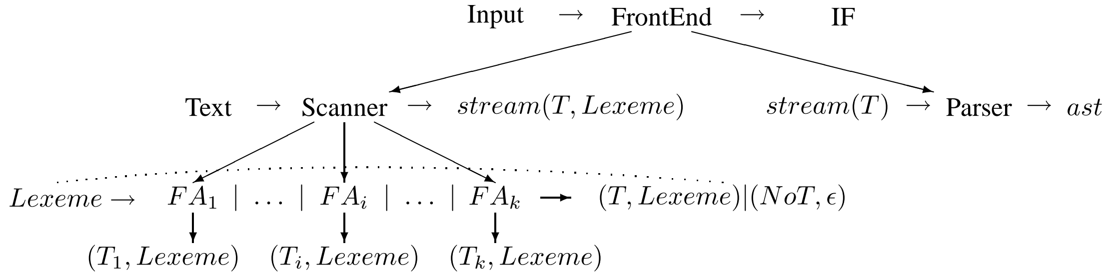

Fig. 4. Front end ontology

The formal definition of this syntax is the object of another research.

The SADL approach to process descriptions is also used in languages such as BPEL [ACD+03], METEOR-S [POV], and WSDL-S [AFM+]. Also, as in WSDL, SADL uses URIs to associate AD ontological terms with the computer artifacts that implement them. However, since SADL operates on the AD ontology, its interpreter does not need the ability to search the web, simply because the URI of its processes are provided in that ontology. In addition, processes in SADL may be composed into larger processes that may be preserved in the AD ontology.

To further explain the motivation behind SADL and the differences from other similar languages, we turn to the problems raised by the implementation of the SADL interpreter. We start by observing that the computation performed by the interpreter relies on composing processes associated with the nodes of the ontology. But these processes are not ready composable because their outputs and inputs may be of different type. Hence, the first problem faced by the IT expert implementing the interpreter consists of finding a universal input/output and standard algorithms that map these input/outputs into this universal form. Our answer to this problem is: *represent the inputs and outputs of the computation processes associated with the nodes of the ontology by XML files and associate the nodes of the ontology with XSLT transformers that perform the mapping of outputs into expected inputs*. The second problem faced by the IT expert is finding a universal way of specifying the address of the computing processes associated with the ontology nodes. Our solution to this problem is: *use URI of theses processes and name spaces associated with various ADs*. The rest is simple compiler technology available to the IT expert. For example, Figure 6 shows the SADL form of the solution algorithm generated by the language expert solving the problem of mapping arithmetical expressions into the values they represent.

```
<sadl>
  <system name="evaluator">
    <component name="scanner"
      location="file://FrontEndOntology/scanner" />
    <component name="parser"
      location="file://FrontEndOntology/parser" />
    <filter name="scanner2parser"
      location="file://FrontEndOntology/scan2parse" />
    <create what="variable" name="input"
      val="stdin" />
    <execute input="input" output="mid"
      process="scanner" />
    <output var="mid" />
    <translate var="mid" filter="scanner2parser" />
    <output var="mid" />
    <execute input="mid" output="output"
      process="parser" />
    <output var="output" />
  </system>
</sadl>
```

Fig. 6. SADL Example

Note that the XML syntax used to express processes in SADL is easily parseable by many pre-existing libraries and tools meaning that SADL modifications can be easily implemented. Because of XML encapsulation the code provides little room for ambiguity and it is easily expandable as creating new attributes for XML elements representing SADL processes does not break older SADL expressions.

C. A domain based problem solving interpreter

The domain based problem solving interpreter (SADLI) is the compositional control element of problem solutions. More specifically, the SADLI is responsible for processing the SADL expressions by carrying out the computational process composition defined by them.

In general the interpreter acts much like the *make* language interpreter (or any scripting language interpreter) in that it provides for assignments, branches, loops, and variable declarations. Variables are abbreviated references to various data

elements present in the AD ontology nodes, including (but not limited to) components, systems, strings and sets which are explained in more detail when discussing the element semantics.

To create a simple example of the SADLI in action, we use a limited number of XML elements and define a few internal data types, namely *string* and *set* which represent strings of characters and sets of variables, respectively. The scope of the XML elements during SADL expression interpretation is handled much like in C where a variable definition exists in the scope where it was defined and all sub-scopes of that scope. The interpreter reserves the variables stdin, stdout and stderr of type *string* to allow for control over input and output to defined processes. The important thing to note about variables in the interpreter context is that rather than creating casting functions like in C/C++, the interpreter calls out to filter functions (XSLT filters, or other component processes) to handle the conversions. These filters are essentially the connectors of the SADL providing a means to glue components together.

The SADLI actions (as seen in Figure 6) are processes specified by the tags of XML elements. This allows SADLI to interpret every XML element that belongs to a SADL expression in a uniform way. The nature of the computation process (defined in the domain ontology or being a SADLI process) is determined by the tag of the element and it is completely specified by the attributes of the element. However, SADL requires some mechanisms to handle control-flow operations such as loops, where certain groupings of SADL processes must be repeated, and branchings, where computation flow performed by some process must be interrupted. To handle this we use the hierarchical nature of XML and thus any SADL expressions between a SADL start and end tag is considered a grouping. The behavior of a grouping depends on the tag and attributes of the tag that surround it. Hence, the tags are used as operators in SADLI and are classified by the nature of the computation they represent as *ontology supported operators* and *SADLI supported operators*. The syntax and the semantics of a few SADLI operators follows:

Ontology supported operators:

1) $< sadl >$ scope $< /sadl >$
   Semantics: informs the interpreter of a SADL definition. While it is somewhat unnecessary, it is a requirement of XML syntax and provides the interpreter with a scope which can contain various SADL expressions.
2) $< component$ name="string" location="URI" $/ >$
   Semantics: creates a variable in the current scope whose name is specified by the attribute *name*. The value of this variable is equivalent to a pointer to the component existing at the location specified by the attribute *location*.
3) $< filter$ name="string" location="URI" $/ >$
   Semantics: creates a variable in the current scope whose name is specified by the attribute *name*. The value of this variable is equivalent to a pointer to the filter existing at the location specified by the attribute *location*. (For more information on filters see *translate*)
4) $< system$ name="string"$>$ scope $< /system >$
   Semantics: defines a new SADL system whose name is specified by the attribute *name*. The scope of the operator *system* is the SADL expression that specifies the system composition.

SADLI supported operators:

1) $< create$ what="string" name="string" type="string" val="string"$/ >$
   Semantic: creates a computing object of nature determined by the attribute *what*. All kinds of computing objects supported by SADL can be created by the operator *create* using an appropriate specifier defined by the attribute *what* and a specific list of attributes required by the creation of the respective computing object. The above syntax uses the attributes for variable and set creation. When a variable is created the name, type, and the value that initializes it are specified by the attributes *name, type, val*. When a set is created the attribute *val* defines a string of comma separated set element specifiers. Further, when *what* determines a process to be created the location of that process is specified by the attribute *location* which is not shown by the above syntax.
2) $< clear$ var="string" $/ >$
   Semantics: removes the variable whose name is specified by the attribute *var* from the scope it was defined in.
3) $< foreach$ element="string" in="string"$>$ scope $< /foreach >$
   Semantics: performs the iteration of the process contained within its scope. The scope is evaluated by the interpreter for each element contained within the set specified by the attribute *in*. At each evaluation the variable with name specified by the attribute *element* is assigned the value of the current element.
4) $< break/ >$
   Semantics: breaks process flow from the current *foreach* scope.
5) $< execute$ input="string" output="string" process="string" $/ >$
   Semantics: instructs the interpreter to execute the process identified by the attribute *process*. The input to the specified process is the contents of the variable specified by the attribute *input*. The results of the execution are stored in the variable specified by the attribute *output*.
6) $< translate$ var="string" filter="filter" $/ >$
   Semantics: instructs the interpreter to perform a filtering on the value of the variable specified by the attribute *var* using the filter specified by the attribute *filter*. The value of the variable should be a XML file. In Acme this filtering action is represented by connectors and in the standard scripting language it is performed by pipes.
7) $< test$ var="string"$>$ scope $< /test >$
   Semantics: provides a run-time test to check if the variable specified by the attribute *var* exists and contains data. If so, the scope is evaluated; otherwise the scope is ignored.
8) $< output$ var="string" $/ >$
   Semantics: instructs the interpreter to output the value of the variable specified by the attribute *var*.

IV. Case-Study

To provide a proof of concept that illustrates our application driven problem solving methodology we present an example

based on the proposed SADL in section 3.2. The problem solved in this case study is: *develop a language interpreter which takes arithmetic expressions as input and returns the values represented by these expressions*. The goal of this example is to illustrate the use of SADL and to demonstrate the ability of current IT tools to implement AD solutions as computational processes.

We make the following assumptions:

- The computational processes used in the SADL expressions are available to SADLI on the local file-system and are limited to executable code on a Linux based system.
- All components take input from standard in (stdin) and output to standard out (stdout).
- The only control flow operator used is the *foreach* operator.
- The result provided by the example interpreter is limited to transient processes and values generated by these processes.

Here we use only two ontology supported processes, a scanner and a parser. The scanner uses a set of regular expressions to gather lexical tokens for integers, reals, and mathematical operations (+,-,*,/). The input to the scanner is a text representing an arithmetic expression and the output is a sequence of tuples $\langle token, value, position\rangle$ representing the lexemes discovered in the input. The parser is much like a standard parser. However, the parser has the ability to map the arithmetic expression into an abstract syntax tree whose leaves are labeled by tokens and their values, and whose interior nodes are operations used in the arithmetic expression. Then the parser walks this abstract syntax tree and evaluates the expression it represents. Thus, the parser takes as input a sequence of tuples $\langle token, value\rangle$, checks that they are the constituents of a valid arithmetic expression, and returns the value of the arithmetic expression thus discovered.

A. Execution walkthrough

In section 3 the AD expert has provided a solution to the above problem using the domain specific language. The equivalent SADL representation of the solution is shown in figure 6. Further, the computing process that implements this solution is performed by the SADLI, as we demonstrate below. The important parts to note here are the sequence of SADL elements with the tags *execute*, *translate*, *execute*. This sequence performs the sequential composition of processes that execute the *scanner*, then translate the scanner output using the *scan2parse* filter, and then execute the *parser*. To demonstrate this, we consider the input
`29 + 4/2`.

The scanner returns an XML file where each element represents a lexical item. The tag of the element is the token of the lexical element it represents, and the attributes *val*, *line*, *word* define the lexeme (value) and its position on the screen. The scanner output for the input example `29 + 4/2` is shown in Figure 7.

```
<stream>
  <int val="29" line="1" word="1" />
  <operator val="+" line="1" word="2" />
  <int val="4" line="1" word="3" />
  <operator val="/" line="1" word="4" />
  <int val="2" line="1" word="5" />
  <newline val="\n" line="1" word="6" />
</stream>
```

Fig. 7. The output

Figure 8 shows the XSLT filter used to translate the file generated by the scanner (Figure 7) into the file in Figure 9 expected by the parser.

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"
   omit-xml-declaration="no" indent="yes" />
<xsl:template match="/stream">
<tokens>
<xsl:for-each select="*">
    <xsl:element name="{name(.)}">
    <xsl:value-of select="@val" />
    </xsl:element>
</xsl:for-each>
</tokens>
</xsl:template>
</xsl:stylesheet>
```

Fig. 8. XSLT filter

There already exists an XSLT translator (xsltproc) that provides this translation functionality. Thus, the *scan2parse* filter is passed, along with the *scanner* output, to the XSLT processor. Figure 9 shows the results of the filter on the scanner output.

```
<tokens>
  <int>29</int>
  <operator>+</operator>
  <int>4</int>
  <operator>/</operator>
  <int>2</int>
  <newline>\n</newline>
</tokens>
```

Fig. 9. Parser input

The parser process takes the result of the XSLT filter as input, generates its internal representation, evaluates it, and deliver a numeric result, as seen in Figure 10. An appropriate filter maps it into what user expects to see.

```
<results>
     <result>31</result>
</results>
```

Fig. 10. Parser output

B. Analysis, Conclusions, and Future Work

Currently the problem solving process is carried out within computer environment irrespective of problem and problem

domain and relies on encoding the problem and its solution algorithm using computer abstractions and concepts. The solution is a machine-language program that run under a given operating system. The research effort to make this process easier for computer user led to the development of a software technology whose complexity becomes unbearable. This paper initiates an alternative methodology whose goal is to move the problem solving process to the problem domain where it belongs. Contrasting this approach with UML we observe that with UML solution algorithms are first represented by UML diagrams which are then manually or automatically transformed into programs of a programming language, such as Fortran. With computational emancipation computer users use domain concepts and abstractions to develop solution algorithms and express them in the natural language of the problem domain. These expressions are then interpreted by custom software which executes the involved computations on computer network by appropriate machines under suitable operating systems. No programming as usual is involved. The complex software meant to support problem solving irrespective of problem and problem domain is replaced by the computational emancipation of the problem domain. This is well illustrated by the great successes in computer usage (such as PowerPoint for research presentation and Tex for printing industry). A new methodology for computer supported teaching evolves and consequently a beneficial feed-back on computational maturation of all fields of human endeavor and a breakthrough in software development are expected. The goodness of this approach cannot be supported by measurements and analysis, simple because there is not yet enough experience with it. The goodness of this approach is however supported by its potential to control software complexity and to simplify computer use, making the computer indeed a problem solving tool dedicated to the problem and problem domain, as it should be.

While the example presented above is very simple, it illustrates the ability to computationally emancipate an application domain and to use this emancipation to develop domain driven software. It is irrelevant to the SADL interpreter whether the components are written in C, C++, Python, Java, or even a shell script. Hence, this work shows that the application domain expert can step away from computer implementations of her problem solutions focusing on the application domain.

Since our evaluator example exists as a *calculator* in the IT domain, there is very little domain to emancipate here. However, these same techniques can be applied in any application domain, and as a future project we will look at this aspect of software development from another domain's point of view. We also need to observe that in our case-study we were both, the domain experts and the IT experts, and thus our component design was well thought out. In the real world, not all components are created equal, in fact, sometimes components provide multiple functionality that must be extracted. In our future work we will look into this aspect of domain oriented component design and will further develop SADL and SADLI to handle the necessary extensions.

REFERENCES

[ACD+03] T. Andrew, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Lymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf, 2003.

[AFM+] R. Akkiraju, J. Farrel, J. Miller, M-T. Naragajan, M.and Schmidt, A. Sheth, and Verma K. Web service semantics - WSDL-S. http://www.w3.org/Submission/WSDL-S/.

[BCM+05] F. Badder, D. Calvanese, D. McGuinnes, D. Nardi, and P. Patel-Schneider, editors. Cambridge University Press, 2005.

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

[Boe96] B. Boehm. Foreword. In *Software Architecture: Perspective on an Emerging Discipline*, pages 1–5. Prentice Hall, 1996.

[CCMWa] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web service description language (last version). http://www.w3.org/TR/wsdl.

[CCMWb] E. Chrstensen, F. Curbera, G. Meredith, and S. Weerawarana. Web service description language (WDSL) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[CM04] M. Crubèzy and M. A. Musen. Ontologies in support of problem solving. In S. Staab and R. Studer, editors, *Handbook on Ontologies*, chapter 16, pages 321–341. Springer, 2004.

[Coo04] S Cook. Domain-specific modeling and model driven architecture. *MDA Journal*, January 2004. Available at: http://www.davidfrankelconsulting.com/MDAJournal.htm.

[FEMR96] D. Fensel, H. Eriksson, M.A. Musen, and Studer R. Conceptual and formal specification of problem-solving methods. *International Journal of Expert Systems*, 9(4):507–532, 1996.

[FM01] D. Fensel and E. Motta. Structured development of problem solving methods. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):913–932, 2001.

[FMvH+03] D. Fensel, E. Motta, F. van Harmelen, V.R. Benjamins, M. Crubèzy, S. Decker, M. Gaspari, R. Groenboom, W. Grosso, M.A. Musen, E. Plaza, G. Schreiber, R. Studer, and B. Weiling. The unified problem-solving method development language upml. *Knowledge and Information Systems*, 5(1):83–131, 2003.

[Fra] D.S. Frankel. MDA journal. http://www.davidfrankelconsulting.com/MDAJournal.htm. Author's email: df@DavidFrankelConsulting.com.

[GBI+04] B. Grady, A. Brown, S. Iyengar, J. Rambaugh, and Selic. B. An MDA manifesto. *MDA Journal*, May 2004. Available at: http://www.davidfrankelconsulting.com/MDAJournal.htm.

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

[GMW00] D. Garlan, R.T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G.T. Leavens and M. Sitaraman, editors, *Foundation of Component-Based Systems*, pages 47–67. Cambridge University Press, 2000.

[POV] A. Patil, A. Oundhakar, S.and Sheth, and K. Verma. METEOR-S web service annotation framework. http://lsdic.cs.uga.edu/lib/download/POSV-WWW2004.pdf.

[RH94] T. Rus and T. Halverson. Algebraic tools for language processing. *Computer Languages*, 20(4):213–238, 1994.

[Rus02] T. Rus. A unified language processing methodology. *Theoretical Computer Science*, 28:499–536, 2002.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice Hall, 1996.

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

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

[vDKV] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. http://www.cwi.nl/arie,paulk,jvisser/.