Application Driven Software for Chemistry

Donald Ephraim Curtis
Member, IEEE
donald-curtis@uiowa.edu
Department of Computer Science
University of Iowa
Iowa City, IA, USA

Teodor Rus
Member, IEEE
rus@cs.uiowa.edu
Department of Computer Science
University of Iowa
Iowa City, IA, USA

Jan Jensen
jhjensen@kemi.ku.dk
Department of Chemistry
University of Copenhagen
Copenhagen, Denmark

***Abstract*—This paper provides a brief introduction to application driven software technology and its usage in the development of a new computer-based problem solving methodology. This new methodology is based on the principle of ”take the computer to the problem” rather than ”taken the problem to the computer” as done by current computer-based problem solving process. We illustrate our research in the Chemistry domain with the problem of finding acid dissociation constants in chemical reactions.**

***Index Terms*—acid dissociation, application domain, chemistry, information technology, ontology, computational emancipation of a problem domain**

I. Introduction

As the demand for computer use in all domains of human endeavor grows, so does the complexity of the software being used. Analysts say that “software complexity is killing IT”[1]. Chemistry is an example of a computer application domain that contains a large amount of complex software tools used for problem solving. Using these tools can be difficult to a point where chemists are forced to become computer experts.

There are two aspects of software complexity. The first, and best studied, concerns the complexity of the software development process. The second, and less studied, concerns software usage in the problem solving process including software maintenance as a process of evolving software to fit the conceptual evolution of the problem domain.This aspect of software complexity is our concern here.

To handle the complexity of software usage during the problem solving process we observe that current computer-based problem solving methodology makes no distinction between various problem domains. In order to use a computer to solve a problem with the current methodology, irrespective of problem and problem domain, the problem-solving algorithm needs to be translated into a program that can then be run on the computer. The unification of potentially infinite many computer application domains, each with potentially infinite many problems, into a “one-size-fits-all” pattern during problem solving is sooner or later meant to “crush” the poor computer. We believe that the only way to prevent the process of “computer-crushing” under the expanding demand for computing required by human-cognition process is to change the one-size-fits-all pattern mentioned above. We seek this change by developing a computer-based problem solving methodology where the computer is “taken to the problem” rather than taking the problem to the computer. In other words, we are developing a computer-based problem solving methodology where the problem solving process takes place *in* the problem domain and the computer is used as a cognitive tool [2]. This methodology has two aspects. The first aspect concerns the process of injecting the computational thinking [3] developed in the information technology (IT) into the application domain (AD). We call this aspect *computational emancipation of application domain*, (CEAD). The second aspect concerns the development of appropriate software tools to support the new problem solving methodology. We call this aspect *application-driven software development* (ADS). This paper is organized as follows: section 2 provides an informal introduction to the CEAD-ing process; section 3 compares ADS software tool usage with conventional software usage; section 4 provides an overview of the ADS software tools; section 5 suggests extensions to the current ADS tools and presents an invitation to experiment with them.

II. Computational emancipation of a problem domain

A computer application domain is characterized by a collection of terms (terminology) and a domain structuring. Domain terms have the same semantic interpretation for all domain experts and are either graphical or textual representations of AD concepts which the domain expert utilizes to formulate problems and develop solution algorithms. We assume that semantics of domain characteristic terms are comput-

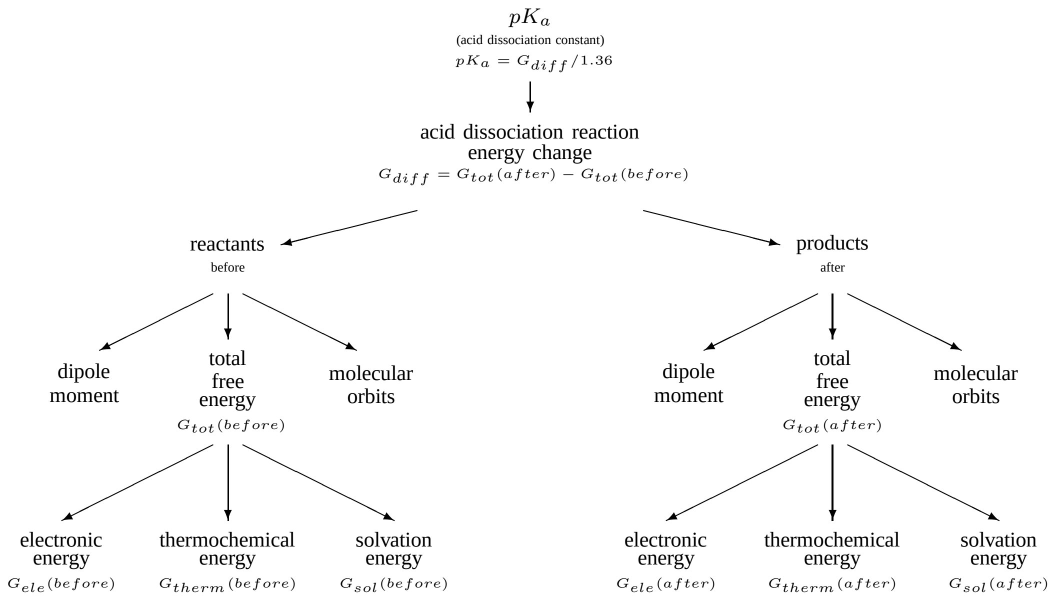

Fig. 1. pK_aOntology

ing objects (such as data and algorithms) that are: universal over the domain, stand-alone, and composable. The domain structuring describes relationships between concepts represented by the domain terminology. We use an ontology [4] to represent the domain structuring and assume that it is incremental. That is, a domain ontology is expanded with terms representing solutions to AD problems. Developing an ontology for an AD means identifying the domain characteristic terms and organizing them into a domain ontology which then becomes a cognitive tool for the problem solving process in the domain.

The process of associating concepts of an AD ontology with IT artifacts implementing them is called *Computational Emancipation of the Application Domain* (CEAD), or what we refer to as the *CEAD-ing the AD* [5], [6]. CEAD-ing the AD brings the IT domain to the AD during the problem solving process. CEAD-ing an AD allows AD experts to manipulate software by manipulating domain concepts using the natural language of the domain. This is the idea behind bringing the computer to the problem. Contrasting the natural language of the domain with the idea of domain specific language as used in current IT technology [7] we observe that while they share many common characteristics they are fundamentally different. Domain specific languages are programming languages whose terms represents instructions to be carried out by a computer. Natural languages of the domain are languages used to represent domain concepts and consequently their terms represent computational processes to be carried out by some computational machinery be it the brain or computer. While a domain specific language expresses a syntactic computation as any other programming language, the natural language of the domain expresses a semantic computation, as any other natural language. But note, natural language of a domain is defined by the domain ontology and consequently it should not be confused with natural language. Further discussion of the natural language of the domain is outside the scope of this paper. As an example of CEAD-ing the AD, in Figure 1 we provide the ontology for computational chemistry related to the problem of *determining the pK_avalue (acid dissociation constant) for a particular acid*.

III. Application driven software development

Application Driven Software Development (ADS) [5], [6] is a methodology that provides the domain expert with a mechanism to solve problems in her domain and execute solution algorithms on the computer without requiring her to translate the solutions into programs. The domain expert sees the computer as a cognitive tool specific to her domain while IT develops software which aides the problem solving process. We demonstrate the ADS approach to problem solving on the chemistry domain. To understand the benefits of using computers as cognitive tools we first describe the process of problem solving following the conventional approach and then illustrate the same process using ADS tools.

```
 $CONTRL SCFTYP=RHF RUNTYP=ENERGY MPLEVL=2 ICHARG=0 $END
 $BASIS GBASIS=N31 NGAUSS=6 NDFUNC=2 NPFUNC=1 DIFFSP=.T. $END
 $GUESS GUESS=HUCKEL $END
 $SCF DIIS=.T. SOSCF=.F. DIRSCF=.T. FDIFF=.F. NPUNCH=1 $END
 $SYSTEM TIMLIM=99999999 MEMORY=30000000 MEMDDI=25 $END
 $DATA
MP2 Electronic Energy
C1
C     6.0    -0.22223     0.00000    -1.37329
C     6.0     0.12774     0.00000     0.08731
O     8.0     1.22570     0.00000     0.53827
O     8.0    -0.95942     0.00000     0.85727
H     1.0    -0.67768     0.00000     1.76702
H     1.0     0.68235     0.00000    -1.96227
H     1.0    -0.81825     0.87505    -1.60518
H     1.0    -0.81825    -0.87505    -1.60518
 $END
```

Fig. 2. Electronic Energy input deck for GAMESS

A. Using GAMESS to compute pK_a

Chemists utilize computers as tools to help compute properties of molecular systems, avoiding the need to perform real-world experiments. We are interested in the problem of finding the *acid dissociation constant for acetate* characterized by the CEAD-ed ontology given in Figure 1. The pK_avalue is useful for, among other things, determining an acid's pH value (level of acidity) and reactivity.

An *acidic dissociation reaction* occurs when an acid is mixed with a base. So let us say we mix *acetate acid* CH_3COOH in water H_2O. The following chemical reaction happens: $H_3COOH + H_20 \rightarrow CH_3COO^- + H_3O^+$. The acetate acid loses a positively charged proton, represented in the equation as a hydrogen atom, and becomes negatively charged (-1). We simplify this reaction to $HA \rightarrow H + A$ where $A = CH_3COO^-$ and H is the positively charged proton.

The pK_avalue can be calculated based on the *energy change* for this reaction, denoted G_{diff}, using the formula: $pK_a = G_{diff}/1.36$. The energy change is the difference between the *total free energy*, denoted G_{tot}, `after` the reaction and `before` the reaction computed by the formulas:

$$\begin{aligned} G_{tot}(before) &= G_{tot}(HA) \\ G_{tot}(after) &= G_{tot}(H) + G_{tot}(A) \\ G_{diff} &= G_{tot}(after) - G_{tot}(before) \end{aligned}$$

G_{tot} for H, A, and HA are the sum of their thermochemical, electronic and solvation energies:

$$G_{tot}(x) = G_{therm}(x) + G_{ele}(x) + G_{sol}(x)$$

The electronic, thermochemical, and solvation energies are values which must be determined experimentally or calculated using universal formulas mathematically proven correct. There are chemistry software packages that perform calculations based on these mathematical formulas, but manipulation of these chemistry packages can be just as complex as developing the software itself. For example, the input for the General Atomic and Molecular Electronic Structure System (GAMESS) [8] developed to calculate the *electronic energy* of acetate acid $G_{ele}(HA)$ is given in Figure 2. The first six lines represent instructions to the GAMESS package to calculate the electronic energy and the data between `C1` and `$END` are coordinates which describe the internal structure of the CH_3COOH molecule. Calculating the pK_avalue for acetate acid requires the chemist to create six *different* input decks like the one in figure 2. Thus, the GAMESS solution to determining the pK_avalue is: solve the problem, learn the language of the GAMESS package, create GAMESS input for all required values, get the result, and calculate the pK_avalue. We observe that while arguably better than doing it by hand, the software complexity of the GAMESS tool makes carrying out a solution a chore.

B. Using ADS to compute pK_a

The idea of application driven software we pursue is to let the chemist identify fundamental concepts she uses while performing problem solving, structure these concepts following their relationships in chemistry, and express solutions in terms of these concepts. Since fundamental terms have already been identified and structured through the ontology in figure 1 the only

```
Acetate_pKa:
  Input: {HA:AcetateAcid, A:Acetate}
  Ouput: AcetateAcidPKA:number
    G_tot(HA) := G_elec(HA) + G_therm(HA) + G_solv(HA)
    G_tot(A) := G_elec(A) + G_therm(A) + G_solv(A)
    G_tot(H) := 0.000 + -4.38 + -262.5
    G_tot(before) := G_tot(HA)
    G_tot(after) := G_tot(A) + G_tot(H)
    G_diff := G_tot(after) - G_tot(before)
    AcetateAcidPKA := G_diff / 1.36
```

Fig. 3. Conceptual Algorithm for Calculating the $pK_a$of Acetate Acid

thing left to do is express the solution in these terms, shown in figure 3.

Note that in order to solve her problem the chemist doesn't have to manipulate any computer artifact. She simple uses the AD ontology to express the solution algorithm using the natural language of the domain and lets the computer manipulate computer artifacts. The conceptual algorithm is executed in the AD by executing *processes* associated with the concepts in the ontology and composing these processes as specified in the conceptual algorithm given by the AD expert.

The ADS solution to determining the pK_avalue is: solve the problem using chemistry concepts in the chemistry ontology and ask ADS software to execute the algorithm and deliver the solution. There is no need for chemist to know *who* executes the algorithms and *how* it is done.

IV. Application Driven Software

The Application Drive Software methodology presented above gives a path for developing tools which manage software complexity and promote problem solving without programming. Modern Graphical User Interfaces (GUI) follow a similar approach by using graphical "widgets" to represent domain concepts. Similar to ADS, these widgets are associated with code that carries out various functionality and thus performing out a solution algorithm is a matter of the user clicking on various parts of the GUI. There are two differences between the approach taken by GUIs and ADS. The first is that composition in a GUI is handled at the code level and thus these domain concepts are not stand alone. Extracting functionality from a GUI for the purpose of utilizing ADS can be a trivial process depending on how heavily the code carrying out the domain process is embedded in the code to draw the GUI. The second difference between GUIs and ADS is at the language level. With ADS one associates code with linguistic terms that represent domain concepts while with a GUI one associates code with iconographic material that may represent domain concepts. In both cases at algorithm execution time an interpreter maps terms or the iconographic symbols into appropriate computer processes. Hence, CEADing can be performed through linguistic or graphical presentations and thus ADS can indeed benefit from utilizing GUI interfaces.

Application driven software developed by us so far consists of:

1) The Software Architecture Description Language (SADL) provided with control flow operators used to represent AD solutions in the IT domain.
2) A translator that takes the conceptual algorithm provided by the domain expert and translates it into a SADL process.
3) A SADL interpreter that performs SADL expressions using computing resources for which the concepts in the ontology are implemented.

These tools provide a means for the ADS methodology and are invisible to AD experts.

The Software Architecture Description Language (SADL) was designed to represent domain level solutions in a machine interpretable form. SADL is not a programming language. Rather, SADL expressions are computer processes that represent AD algorithms using XML syntax so they can more easily be carried out by a computer. Execution is handled by the SADL interpreter which initiates and controls the execution of the computer artifacts associated with the concepts of the ontology used in the solution algorithm. Semantic elements of SADL are processes, process composition operators, and systems. Processes are represented by attributed files. A process's attributes specify process location, action the process performs (if any), input/output, etc. Among others, the following are attributes used in SADL XML elements to specify processes:

name - The name attribute identifies the domain concept which is to be executed for the problem solution.

```
<?xml version="1.0" ?>
<sadl>
  <system name="acetate_pka" input="uri(HA) uri(A)" output="uri(AcetateAcidPKA)">
    <!-- Acetate Acid -->
    <component name="uri(G_elec)" input="uri(HA)" output="uri(G_elec(HA))" />
    <component name="uri(G_solv)" input="uri(HA)" output="uri(G_solv(HA))" />
    <component name="uri(G_therm)" input="uri(HA)" output="uri(G_therm(HA))" />
    <component name="uri(+)"
      input="uri(G_therm(HA)) uri(G_solv(HA)) uri(G_elec(HA))"
      output="uri(G_tot(HA))" />
    <!-- Acetate -->
    <component name="uri(G_elec)" input="uri(A)" output="uri(G_elec(A))" />
    <component name="uri(G_solv)" input="uri(A)" output="uri(G_solv(A))" />
    <component name="uri(G_therm)" input="uri(A)" output="uri(G_therm(A))" />
    <component name="uri(+)"
      input="uri(G_therm(A)) uri(G_solv(A)) uri(G_elec(A))"
      output="uri(G_tot(A))" />
    <!-- Proton -->
    <component name="uri(+)" input="0.000 -4.38 -262.5" output="uri(G_tot(H))" />
    <!-- Before -->
    <component name="uri(null)" input="uri(G_tot(HA))" output="uri(G_tot(before)" />
    <!-- After -->
    <component name="uri(+)" input="uri(G_tot(H)) uri(G_tot(A))" output="uri(G_tot(after))" />
    <!-- Difference -->
    <component name="uri(-)" input="uri(G_tot(after)) uri(G_tot(before))" output="uri(G_diff)" />
    <!-- pKa -->
    <component name="uri(/)" input="uri(G_diff) 1.36" output="uri(AcetateAcidPKA)" />
  </system>
</sadl>
```

Fig. 4. SADL Expression of Acetate Acid pK_aSolution

input - The input attribute gives a space separated list of concepts which are to be used for input to the domain concept being executed.

output - The output attribute identifies the concept being returned.

Using ADS tools and the CEAD-ed chemistry ontology, the solution given in figure 3 is mapped into the SADL expression given in Figure 4. Computer artifacts associated with concepts in the CEAD-ed ontology are named using Uniform Resource Identifiers [9]. To simplify the SADL expression we have replaced the URI for each `concept` with the expression `uri(<concept>)`. Once mapped into a SADL expression, the solution is carried out by the SADL interpretor by performing the processes existing at each URI. The result is the pK_avalue for acetate acid. Note that this is not a simple interpretation process as performed by the interpreters of current technology. The symbols used in the domain algorithm represent concepts of the domain. For example, the symbols '+', '-', etc., may not be the arithmetical operations because they represent the concepts which problem domain denotes by these symbols. The ADS manner of algorithm execution is better compared with the web-service execution process [10].

Through *computational emancipation of the application domain* the process of converting domain level solutions to SADL expressions is trivial. Every concept used in the domain algorithm is searched in the ontology and is replaced by an XML element whose tag is the concept itself and the attributes are determined from the associated computer artifact. The control flow operations are replaced by the appropriate XML elements representing process composition operators. The resulting XML expression represents the computer artifact implementing the concept defined by the domain algorithm, and thus extends the domain ontology while preserving computational emancipation of the application domain.

The interpretor used for this work is available at `http://www.cs.uiowa.edu/˜dcurtis/sadl/` and is ready for experimentation. We appreciate any observations that would help us to further extend and improve it.

V. Conclusion

Application Driven Software (ADS) bridges the conceptual gap between IT and the application domain and provides a problem solving process that is characterized by "hands on the problem" rather than "hands on the computer". This opens the door to new patterns of collaborations. It also shows the way toward developing solutions that handle software complexity *without generating more software complexity*.

The work presented in this paper demonstrates that ADS is ready for use. But as expected, this raises new research problems. Among others, the problem of synthesizing the *natural language of the domain*, to be used by domain experts while expressing solution algorithms, is of a particular interest for further developments. Currently we are developing a mechanism to extract this language from the CEAD-ed domain.

References

[1] P. Krill, “Complexity is killing it, say analysts,” Techworld: www.techworld.com.

[2] S. Lajoie and S. Derry, Eds., *Computers as Cognitive Tools I*. Hillsdale, Lawrence Erlbaum Associates, Inc., 1993.

[3] J. Wing, “Computational thinking,” *Communications of the ACM*, vol. 49, no. 3, pp. 33–35, 2006.

[4] N. Noy and D. McGuinness, “Ontology development 101: A guide to creating your first ontology,” http://ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness-abstract.html, Stanford University, Stanford, CA 94305.

[5] T. Rus and D. Curtis, “Application driven software development,” in *International Conference on Software Engineering Advances, Proceedings*, Tahiti, 2006.

[6] ——, “Toward application driven software technology,” in *The 2007 World Congress in Computer Science, Computer Engineering, and Applied Computing,WORLDCOMP’07*, Las Vegas, USA, 2007.

[7] J. Heering and M. Mernik, “Domain-specific languages for software engineering,” in *Proceedings of the 35th Annual Hawaii International Conference on System Science*, 2002.

[8] G. R. Group, “The general atomic and molecular electronic structure system (GAMESS),” http://www.msg.ameslab.gov/GAMESS/.

[9] B.-L. T., R. Fielding, U. Irvine, and L. Masinter, “RFC 2396: Uniform resource identifiers (URI): Generic syntax,” http://www.ietf.org/rfc/rfc2396.txt, 1998.

[10] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarane, “Business process execution language for web services,” http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.