A Language-centric Approach to Software Engineering: Domain Specific Languages meet Software Components

Gopal Gupta

Department of Computer Science
University of Texas at Dallas
[gupta@cs.utdallas.edu]

Abstract. Domain Specific Languages (DSLs) are high level languages designed for solving problems in a particular domain, and have been suggested as means for developing reliable software systems. Component based software engineering has been proposed as a way of reducing the complexity of software development by providing ready-made *software components* for parts of the system being developed. We present an approach that amalgamates these two technologies. Our approach relies on a (constraint) logic programming-based framework for specification, efficient implementation, and automatic verification of domain specific languages (DSLs) around software components. Our framework allows the implementation infrastructure for a DSL (interpreter, compiler, debugger, and profiler) to be automatically obtained (in a provably correct manner) from the semantic specification of the DSL. Additionally, the semantic specification can be used for (semi-)automatically verifying programs written in the DSL as well as for automatically checking that *component contracts* are consistent with the manner in which they are used. This new framework is currently being applied for developing several DSLs, designed around software components. The most significant of these is a DSL, called ΦLog, being developed to enable biologists to program phylogenetic problems in biology.

1 Introduction

Writing software that is robust and reliable is a major problem that software developers and designers face today. Development of techniques for building reliable software has been an area of study for quite some time. Recently, two distinct approaches have been proposed:

– **Approaches based on domain specific languages (DSL):** In the DSL approach [5, 25, 14, 20, 22, 13], a domain specifi c language is developed to allow users to solve problems in a particular application area. A DSL allows users to develop complete application programs in a particular domain. Domain specifi c languages are very high level languages in which domain experts can write programs *at a level of abstraction at which they think and reason*. DSLs are not "general purpose" languages, rather they are supposed to be just expressive enough to "capture the semantics of an application domain" [20]. The fact that users are able to code problems at the level of abstraction at which they think and the level at which they understand the specific application domain results in programs that are more likely to be correct, that are easier to write, understand and reason about, and easier to maintain. As a net result, programmer productivity is considerably improved. The DSL-based approach can be regarded as a top-down approach, in which a simple, high-level language interface is presented to the programmer to ease the task of programming.

– **Approaches based on Component Based Software Engineering (CBSE)** In this approach [2–4, 29] a repository of ready-made software components is assumed. Programmers write "glue code" to put together existing software components to solve a particular problem. The "glue code" is written in a traditional programming language and makes use of components that can be regarded as ready-made library procedures. Use of components results in software-reuse: tasks that are similar in nature need not be programmed again and again. The components based approach also results in improved programmer productivity due to software reuse. If the library of components is standardized, the components based approach can also result in code that is easier to read and maintain (e.g., components can be regarded as plug-in modules whose implementation can change, and as long as the interface of the component remains fixed, the software that uses components need not be changed). The components-based approach can be thought of as a bottom-up approach in which low level implementation details of those parts of the program—for which components are available—can be hidden from the programmer.

Both DSL-based and CBSE-based approaches have their advantages as well disadvantages. The two technologies of DSL and components can be synergistically used to create software faster and in a provably correct manner. We argue that combining the two overcomes some of the disadvantages of both approaches. We achieve this combination via a semantics-based approach that yields an efficient implementation infrastructure (interpreters, compilers, debuggers, and profilers) for DSLs. In this semantics-based approach, a *denotational semantics* [27] of the DSL is written in terms of software components available for that application domain (components are, in fact, treated as part of the *semantic algebras* [27] of this semantics). This semantics is coded using *Horn Logic* (or pure Prolog) and *Constraints* [28], and is executable [17]. The executable semantics yields an interpreter that may make calls to the various components. The executable semantics can be extended in a simple way to obtain debuggers and profilers. Given a program P written in the DSL, the semantic interpreter of the DSL can be partially evaluated [21] w.r.t. P to obtain "compiled code" in terms of calls to software components. Because the interpreter, compiler, debugger and profiler are all derived from the semantic specification, they are provably correct and obtained automatically from the semantic specification of the DSL. Also, the time taken for each iteration of the DSL design is much less, as changing the DSL only requires making change to its semantics, the modified implementation infrastructure (i.e., the interpreter, compiler, debugger, profiler) can be automatically derived from this semantics. Defining the semantics of DSL in terms of components, makes the task of specifying this semantics easier. Finally, component contracts can be specifi ed as constraints [15], their consistency w.r.t. components' use can be checked in the semantic interpreter.

2 Domain Specific Languages (DSL)

The task of developing a program to solve a specifi c problem involves two steps. The fi rst step is to devise a solution procedure to solve the problem. This steps requires a domain expert to use his/her domain knowledge, expertise, creativity and mental acumen, to devise a solution to the problem. The second step is to code the solution in some executable notation (such as a computer programming language) to obtain a program that can then be run on a computer to solve the problem. In the second step the user is required to map the *steps* of the solution procedure to *constructs* of the programming language being used for coding. Both steps are cognitively challenging and require considerable amount of thinking and mental activity. The more we can reduce the amount of mental activity involved in both steps (e.g., via automation), the more reliable the process of program construction will be. Not much can be done about the fi rst step as far as reducing the amount of mental activity is involved, however, a lot can be done for the second step. The amount of mental effort the programmer has to put in the second step depends on the "semantic" gap between the level of abstraction at which the solution procedure has been conceived and the various constructs of the programming language being used. Domain experts usually think at a very high level of abstraction while designing the solution procedure. As a result, the more low-level is the programming language, the wider the semantic gap, and the harder the user's task. In contrast, if we had a language that was right at the level of abstraction at which the user thinks, the task of constructing the program would be much easier. A domain specifi c language indeed makes this possible.

However, a considerable amount of infrastructure is needed to support a DSL, a major disadvantage of this approach. First of all, the DSL should be manually designed. The design of the language will require the inputs of both computer scientists and domain experts. Once the DSL has been designed, we need a program development environment (an interpreter or a compiler, debuggers, editors, etc.) to facilitate the development of programs written in this DSL. The implementation infrastructure of the DSL (i.e., its compilers, debuggers, profi lers, etc.) will constantly change as the language evolves. Making changes to the implementation infrastructure is a daunting task, and we believe is a major hurdle to the DSL-based approach being widely employed. Leveson et al [22] have used the DSL based approach for designing software for airplane control. However, they observe that the design of the DSL can take as much as 3 years [22]. We believe that this is primarily because of the reason that a language is fully understood only after its implementation infrastructure (interpreter, compiler, etc.) has been developed and used for writing and executing a few programs by the domain experts. Developing the implementation infrastructure, or modifying and changing it takes a long time, resulting in each design iteration of the DSL taking a long time as well.

3 Software Components

A software component [29, 2–4] is a unit of independent deployment that has no persistent state and that may have been developed by a third party. Software components can be thought of as software modules that have been developed for commonly encountered tasks and that can be employed in any software system when needed. Components have contractually specifi ed interfaces and explicit context dependencies only [29]. The advantage of software components is they can be bought from third party, and can be freely reused. Repositories of software components have been developed for use in software development projects to reduce the programming effort involved. The main advantage of software components is they facilitate software reuse, and thus can considerably reduce programmers' burden. In the software components based approach components are composed together using "glue code" written in some traditional language.

The CBSE approach frees a programmer from reprogramming many tasks. However, many problems still arise or remain. CBSE does not completely free the programmer from low-level programming since the components still have to be glued together in a low-level way. Essentially, all tasks in a software system for which components cannot be found still have to be programmed in a low level way. Finding components from a component repository that are suited to one's software needs is also a diffi cult task as component repositories can be quite large. Also, in the CBSE approach system integration has to be brought to the forefront of the software development process (typically it's at the end of the software development phase) and continually managed. The hardest problem in the CBSE approach, we believe, is knowing which components to use in a system, since pre-existing set of components may have been written for a pre-existing, possibly unknown, set of requirements (specifi ed in component contracts [29]). These requirements may be very general, in which case the requirements of the system to be built will have to be made to conform to these general requirements, or the requirements with which the component is written may be quite restrictive and may fundamentally conflict with the requirements of the system in which this component is needed.

4 Domain Specific Languages and Software Components

In this position paper we espouse an approach that makes use of both DSLs and software components thereby producing a framework in which, we believe, software can be developed faster and more reliably. Our thesis is that components should be embedded in a domain specifi c language once and for all and this DSL should then be used by developers for writing applications. This is in contrast to having developers use components directly in their applications. The DSL can be thought of as a high-level wrapper language built around a set of components that are likely to be used in an application domain. The task of matching the requirements of the components and their suitability becomes the responsibility of the DSL designer(s) during language design, rather than of every application developer who uses components. Essentially, the language designer provides a proper abstraction for the components in the DSL itself in a cohesive way, freeing the developer from having to deal with the vagaries of a component's interface. The application developer only has to master the DSL, and thus avoids having to struggle with understanding the component's interface.

We have developed a semantics-based approach for combining DSL with components which works as follows. An application domain in which problems need to be solved is identifi ed. A high-level domain specifi c language is designed so that domain experts can write applications at their level of abstraction. The semantics of this domain specifi c language is denotationally specifi ed in terms of software components available for that domain. These software components have to be identifi ed by the language designer(s). The semantics is coded using Horn Logic (pure Prolog) [28, 17] and is executable. The *executable semantics automatically yields an interpreter for the DSL* (this interpreter will call the various components during execution of a DSL program). The executable semantics can be extended in a simple way to obtain debuggers and profi lers [19]. This is possible because the declarative semantic of the DSL explicitly passes the execution state as an argument of a predicate; hooks can be added after each call to semantic predicates that cause execution to pause and exhibit the execution state just as a debugger would. Likewise hooks can be added that record and compile execution statistics just as a profi ler would. Given a program P written in the DSL, the semantic interpreter of the DSL can be partially evaluated [21] w.r.t. P (using partial evaluators for Prolog such as Mixtus [26]) to obtain "compiled code" in terms of calls to these software components [17, 15].

A complete description of the framework as well as examples illustrating our approach [17, 19, 18] can be found elsewhere and are not included here due to lack of space. Our approach is currently being applied to develop a domain specifi c language called Φ-log [24] for allowing biologists to program phylogenetic applications. Φ-log is being implemented on top of software components developed by computational biologists for solving problems in phylogenetics. Our approach also being applied to develop a DSL for e-commerce applications.

5 Advantages of the Framework

Our framework combining DSLs and CBSE eliminates many of the disadvantages associated with software components and DSLs. The availability of components makes the design and specifi cation of DSL much faster. The DSL in turn acts as a high-level "wrapper" around software components, freeing individual application developers from having to worry about potential mismatches between the interface of software components and the applications.

The principal advantage of combining DSL and software components via a semantics-based approach is that that the implementation infrastructure (interpreter, compiler, debugger, profi ler) for a DSL can be rapidly prototyped. This can considerably speed up the iterative design-implement-modify-reimplement cycle involved in DSL design, thus removing what we believe to be one of the major hurdles that has precluded widespread use of DSL technology.

A semantic-based approach also facilitates development of DSLs based on software components available in a particular domain. Thus, a software engineering expert can look at a components repository, identify a set of software components in a particular domain and then design a DSL around these components to facilitate programming of tasks in that domain. The Horn logical semantics of the DSL will be written in terms of this set of components to obtain the implementation infrastructure for this DSL.

Thus, our approach can be applied in two ways: (i) in a top down manner, where an application domain is identifi ed, a DSL designed, and its implementation infrastructure obtained around a set of software components using our semantics based framework; or, (ii) in a bottom up fashion, i.e., a set of closely related software components in a large repository is identifi ed (in consultation with the users), and then a DSL designed around these software components, and its implementation infrastructure obtained. Note that in both cases, identifi cation of the components and the design of the DSL will be collaboratively done by computer experts and the domain experts, however, the implementation infrastructure will be developed by the computer expert alone.

Once the design of a DSL has been fi xed, and its implementation prototyped, highly effi cient implementations can be obtained by implementing optimizing compilers for the DSL using the traditional compiler technology.

6 Language-centric Software Engineering

It can be argued that any complex software system that interacts with the outside world defi nes a domain specifi c language. This is because the input language that a user uses to interact with this software can be thought of as a domain specifi c language. For instance, consider a fi le-editor; the command language of the fi le-editor constitutes a domain specifi c language. This language-centric view can be quite advantageous to support the software development process. This is because the *semantic specification of the input language of a software system is also a specification of that software system*—we assume the semantic specifi cation also includes the syntax specifi cation of the input language. *If the semantic specification of the input language is executable, then we obtain an executable specification of the software system*. Note that this semantic specifi cation can be given in terms of software components (i.e., s/w components are treated as primitive operations in the semantic specifi cation). The preceding observations can be used to design a language semantics based framework for specifying, (effi ciently) implementing using s/w components, and verifying (rather model checking or debugging in a structured way) software systems. The syntax and semantics of the input language is specifi ed using Horn logic/Constraints, and is executable. Effi cient, provably correct, compilation of the software systems in terms of software components can be obtained via partial evaluation. The resulting executable specifi cation can also be used for verifi cation, model checking and structured debugging. Thus, in the light of the above discussion, software design is seen as the task of designing an input language. Implementation is seen as specifying the semantics of this input language in terms of software components.

An obvious candidate framework for specifying the semantics of a domain specifi c language is denotational semantics [27]. Denotational semantics has three components: (i) syntax, which is typically specifi ed using a BNF, (ii) semantic algebras, or value spaces, in terms of which the meaning is given (in our framework software components will be treated as a semantic algebras), and, (iii) valuation functions, which map abstract syntax to semantic algebras. In traditional denotational definitions, syntax is specified using BNF, and the semantic algebra and valuation functions using λ-calculus. There are various practical problems with the traditional approach: (i) the syntax is not directly executable, i.e., it does not immediately yield a parser, (ii) the semantic specification cannot be easily used for automatic verification or model checking. Additionally, the use of separate notations for the different components of the semantics implies the need of adopting different tools, further complicating the process of converting the specification into an executable tool. Verification should be a major use of any semantics, however, this has not happened for denotational semantics; its use is mostly limited to studying language features, and (manually) proving properties of language constructs (e.g., by use of *fixpoint induction*). In our framework, we use Horn logic (or pure Prolog) for expressing denotational semantics as this facilitates the specification, implementation, and automatic verification/debugging of DSL programs, all in one framework. In the Horn logical denotational semantics framework, the BNF grammar can be specified as a *definite clause grammar* (syntax specification), which automatically yields a parser (executable syntax). The semantic algebras are defined in terms of software components and pure Prolog while the valuation functions (rather valuation predicates) are defined using pure Prolog. The valuation predicates can be executed on a Prolog interpreter yielding an executable semantics.

7 Applications of the Framework

Our logic programming based approach to software engineering is being applied to solve a number of problems. We are currently designing a domain specific language to enable biologists to program solutions to *phylogenetic inference problems* [24]. Phylogenetic inference [10] involves study of the biocomplexity of the environment based on genetic sequencing and genetic matching. Solving a typical problem requires use of a number of software systems—Genbank [6] (a repository of genetic data), BLAST [7] (a program for querying genetic databases), CLUSTAL W [8] (a program for aligning molecular sequences), PHYLIP [12] and PAUP [9] (both are programs for inferring evolutionary paths), etc., along with a number of manual steps (e.g., judging which sequence alignment for two genes is the "best"), as well as extra low-level coding to glue everything together. A biologist has to be considerably sophisticated in use and programming of computers to solve these problems, as outputs from various software systems have to be massaged and transformed, and then fed to other software systems. We are developing a DSL for phylogenetic inference that will allow biologists to program such interactions at a very high level, essentially allowing them to write/debug/profile programs at their level of abstraction. The semantic specification of this DSL is given in terms of available software components for phylogenetic inference (Genbank, CLUSTAL W, BLAST, PHYLIP, PAUP, etc). The task of solving phylogenetic problems will become much simpler for the biologist, giving them the opportunity to become more productive as well as be able to try out different "what-if?" scenarios.

Our approach is also being used to facilitate the navigation of complex web-structures (e.g. tables and frame-based pages) by blind users (blind-users typically access the

WEB using audio-based interfaces). Given a complex structure, say a table, the web-page designer may wish to communicate only the essential parts of the table to a blind-user. In our approach, the web page-writer (or a third party) will attach to the web-page a domain specifi c language program that encodes the table navigation instructions [23].

Finally, our approach is also being used to generate provably correct code for SCR [1] specifi cations, and for developing a domain specifi c language for writing e-commerce applications.

8 Formal Verification

Automated or semi-automated verifi cation is also possible in our semantics-based framework. Component contracts can also be enforced in the same framework. The semantic specifi cation of a DSL $\mathcal{L}$ coded in Horn logic can be viewed as an *axiomatization* of the language constructs of $\mathcal{L}$. The denotation of a program $\mathcal{P}_L$ written in $\mathcal{L}$ w.r.t. the Horn logical semantics of $\mathcal{L}$, $\mathcal{P}_d$, can be thought of as an axiomatization of the logic implicit in the program $\mathcal{P}_L$ or as an axiomatization of the problem that $\mathcal{P}_L$ is supposed to solve. This axiomatization can be used in conjunction with a logic programming engine to perform verifi cation. Additionally, the relational nature of logic programming allows for the state space of a program written in $\mathcal{L}$ to be explored with ease. This fact can also be exploited to debug/verify properties of programs.

One standard way of gaining more confi dence is to prove interesting properties about $\mathcal{P}_L$ but instead of using $\mathcal{P}_L$ we use its Horn logical denotation, $\mathcal{P}_d$, instead. Thus, given a property Φ that we want to prove about $\mathcal{P}_L$, we show that Φ is a logical consequence of axioms in $\mathcal{P}_d$, i.e., $\mathcal{P}_d \models \Phi$. However, note that the program may have been written under certain assumptions that were made regarding the input to the program.

Let us assume that we have a precondition I on inputs, $\bar{X}$, and a postcondition O on outputs, $\bar{Y}$, of the program $\mathcal{P}_L$. This means that for program $\mathcal{P}_L$, if $I(\bar{X})$ is true and $\mathcal{P}_L(\bar{X})$ terminates and produces outputs $\bar{Y}$, then $O(\bar{Y})$ must be true if $\mathcal{P}_L$ is correct. Let us assume that $\texttt{main\_p}(\bar{X}, \bar{Y})$ represents $\mathcal{P}_L$'s Horn logical-denotation. Then, the formula

$$(\mathcal{I}(\bar{X}) \wedge \texttt{main\_p}(\bar{X}, \bar{Y}) \rightarrow \mathcal{O}(\bar{Y})) \tag{1}$$

must hold true. Alternatively, the formula

$$(\mathcal{I}(\bar{X}) \wedge \texttt{main\_p}(\bar{X}, \bar{Y}) \wedge \neg\mathcal{O}(\bar{Y})) \tag{2}$$

must be false. We can use the formula above as a query to an LP system on which the logic program $\mathcal{P}_d$ has been loaded. If the program is correct (i.e., program terminates and the postconditions hold) then the above query would fail. Note that if components are being used as part of the semantic algebra in defi ning the semantics, then preconditions and postconditions that component satisfi es may have to identifi ed for each component and coded as logic/constraint programs. It is customary to specify *component contracts* via preconditions and postconditions. These preconditions and postconditions will be used during the verifi cation of the program that employs these components in the above query.

If component contracts are specifi ed as constraints, i.e., both preconditions and postconditions of a component $\mathcal{C}$ are specifi ed as constraints, then these preconditions can be directly inserted into the denotation of the program that uses $\mathcal{C}$. If these constraints are consistent with the rest of the program denotation, then the constraints comprising the postconditions can be conjoined with the program denotation to complete the consistency proof. Note that use of constraints and logic programs for enforcing component contracts have also been advocated by others [15, 11], however, our approach embeds them in a semantics-based framework.

9 Conclusion

In this position paper we presented a framework that combines on domain specifi c languages with software components. In this framework, Domain Specifi c Languages are built around software components for rapid design, and their implementation infrastructure rapidly realized using a semantics based approach. The semantics based approach also permits automatic verifi cation as well as generation of provably correct code. Note that the semantics and logic programming based approach is transparent to the end-user, i.e., the end-user only writes a program in the DSL, and is completely unaware of the underlying implementation technology used. We believe that using the semantics and logic programming based approach DSLs can be rapidly designed around collections of components in a particular domain, reducing the cost of prototyping and software development.

Acknowledgments The authors wish to thank Neil Jones, Enrico Pontelli, and Kishore Kamarupalle for helpful input. The author has been partially supported by grants from the NSF.

References

1. C. L. Heitmeyer, R. Jeffords, B. Labaw. Automated Consistency Checking of Requirements Specification. ACM TOSEM 5(3):231-261. 1996.
2. K. Bergner, A. Rausch, M. Sihling. Componentware—the Big Picture. 20th ICSE Workshop on Component-based Software Engineering. 1998.
3. M. Zaremski, J. M. Wing. Specification Matching of Software Components. ACM TOSEM. 6(4):33-369. 1997.
4. R. Schmidt and U. Assman. "Concepts for developing components-based systems." 20th ICSE Workshop on Component-based Software Engg., Japan, 1998.
5. J. Bentley. Little Languages. *CACM*, 29(8):711-721, 1986.
6. GenBank Overview. www.ncbi.nlm.nih.gov/Genbank.
7. S.F. Altschul and B.W. Erickson. Significance of nucleotide sequence alignments: a method for random sequence permutation that preserves dinucleotide and codon usage. *Mol. Biol. Evol.*, 2:526–538, 1985.
8. D. G. Higgins, J. D. Thompson, and T. J. Gibson. Using CLUSTAL for multiple sequence alignments. *Methods in Enzymology*, 266:383–402, 1996.
9. D. L. Swofford. PAUP: phylogenetic analysis using parsimony, version 3.1.1. TR, Illinois Natural History Survey, 1993.

10. D.L. Swofford, G.J. Olsen, P.J. Waddell, and D.M. Hillis. Phylogenetic inference. In David M. Hillis, Craig Moritz, and Barbara K. Mable, editors, *Molecular Systematics*, chapter 11, pages 407–514. Sinauer, Sunderland, MA, second edition, 1996.
11. A. Cernuda del Rio, J. E. Labra Gayo, J. M. Cueva Lovelle. Applying the Itacio Verification Model to a Component-based Real-Time Sound Processing System. *Proc. of 2nd International Workshop on Logic Programming and Software Engg.*, 2001, Paphos, Cyprus.
12. J. Felsenstein. PHYLIP: Phylogeny inference package, version 3.5c, 1993. see http://evolution.genetics.washington.edu/phylip/software.html, 2000.
13. W. Codenie, K. De Hondt, P. Steyaert and A. Vercammen. From custom applications to domain-specific frameworks. In *Communications of the ACM*,Vol. 40, No. 10, pages 70-77, 1997.
14. C. Consel. Architecturing Software Using a Methodology for Language Development. In *Proc. 10th Int'l Symp. on Prog. Lang. Impl., Logics and Programs (PLILP)*, Springer LNCS 1490, pp. 170-194, 1998.
15. S. M. Daniel. An Optimal Control System based on Logic Programming for Automated Synthesis of Software Systems Using Commodity Objects. *Proc. Workshop on Logic Prog. and Software Engg*. UK, July 2000.
16. L. King, G. Gupta, E. Pontelli. Verification of a Controller for BART: An Approach based on Horn Logic and Denotational Semantics. In *High Integrity Software Systems*. Kluwer Academic Publishers.
17. G. Gupta. Horn Logic Denotations and Their Applications. In *The Logic Programming Paradigm: The next 25 years*, Proc. Workshop on Strategic Research Directions in Logic Prog., LNAI, Springer Verlag, May 1999.
18. G. Gupta. Logic Programming based Frameworks for Software Engineering. *Proc. Workshop on Logic Programming and Software Enginering*. London, UK, July 2000.
19. G. Gupta and E. Pontelli. Specification, Implementation, and Verification of Domain Specific Languages: A Logic Programming-based Approach. Essays in honor of Bob Kowalski. Spriger Verlag LNAI 2407, 2001, pp. 211-239
20. P. Hudak. Modular Domain Specific Languages and Tools. In *IEEE Software Reuse Conf*. 2000.
21. N. Jones. Introduction to Partial Evaluation. In *ACM Computing Surveys*. 28(3):480-503, 1996.
22. N. G. Leveson, M. P. E. Heimdahl, and J. D. Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Software Engineering - ESEC/FSE*, Springer Verlag, pages 127-145, 1999.
23. E. Pontelli, W. Xiong, G. Gupta, A. Karshmer. *A Domain Specifi c Language Framework for Non-Visual Browsing of Complex HTML Structures* ACM Int. Conference on Assistive Technologies, 2000.
24. E. Pontelli, D. Ranjan, G. Gupta, and B. Milligan. ΦLog: A Domain Specific Language for Describing Phylogenetic Inference Processes. In *Proc. First IEEE Computer Society Bioinformatics Conference*. Aug. 2002. IEEE Press, Los Alamitos, CA.
25. C. Ramming. *Proc. Usenix Conf. on Domain-Specifi c Languages.* Usenix, 1997.
26. D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* Ph.D. Thesis, Royal Inst. of Techn. Sweden, 1994.
27. D. Schmidt. *Denotational Semantics: a Methodology for Language Development*. W. C. Brown Publishers, 1986.
28. L. Sterling and S. Shapiro. *The Art of Prolog.* MIT Press, 1996.
29. C. Szyperski. Component Software: Beyond Object-oriented Programming ACM Press, New York. 1998.