Language Based Software Engineering

Gopal Gupta

Applied Logic, Programming Languages and Systems Lab
Department of Computer Science
University of Texas at Dallas
Richardson, TX, USA 95083
gupta@utdallas.edu

The three most important things in software development: notation, notation, notation.

1 Domain Specific Languages: Software as a Language Processor

Techniques for developing reliable, error free (and these days also secure) software have been a subject of research for a long time. A number of software engineering frameworks have been proposed, from the *waterfall model* [1] to *aspect oriented frameworks* [2]. In this short paper we present a language-based framework for software engineering [10]. In the language-based software engineering framework domain experts are the application developers. They program their applications in high level *domain specific* languages. This approach, based on using high level DSLs, results in more robust, reliable and secure software.

The task of developing a program to solve a specific problem involves two steps. The first step is to devise a solution procedure to solve the problem. This step 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 software construction will be. Not much can be done about the first 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 procedures. As a result, the more low-level 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 specific language indeed makes this possible.

Note that for any kind of communication (including between programmers/users and the computer) the notation used for the communication determines how effectively and efficiently that communication can be understood/processed. A new, complex notation may take longer time to learn, but if the notation is carefully designed for the task/domain, then it allows far more complex things to be expressed and understood considerably more easily. It also encourages creativity, since people now have to simply focus on what to say, rather than on how to say it. This in turn affects productivity of the people (and machines) involved. A prime example of this is the position-based decimal notation for arithmetic, which allows for efficient algorithms for various arithmetic operations (addition, subtraction, multiplication, etc.) to be developed. Imagine developing a method similar to long division for the Roman numeral notation! The importance of notation cannot be overstated for programming. The proliferation of programming languages has come about due to our desire to find the "right" language that will make it easy for programmers to express their algorithms (of course, the choice of algorithm some times depends on the notation being used, but here I use the term algorithm in a more general sense, namely, as a method for solving a problem). Since there can be no one perfect notation for all tasks, a plethora of programming languages have been developed, each one optimized for one particular type of use. One could take this proliferation to its logical conclusion and argue that one needs a dedicated (domain specific) programming language for each type of task and domain.

In the DSL based approach to software engineering, software engineers first design a domain specific language for the tasks at hand. The implementation of the software system then amounts to implementing a language processor (interpreter/compiler) for this domain specific language. End-users (who are usually domain experts) then write programs using this domain specific language. Dually, any software system can be viewed as a language processor. This is because any complex software system can be understood in terms of how a user will interact with it. A user interacts with a software by providing input data and commands. Thus, to understand a software system, one has to understand the input language used to specify these data and commands. This input language, of course, can be regarded as a domain specific language. The software system can be thought of as imparting operational semantics to this input language. Thus, the task of developing a complex software system can be thought of as first designing its input language, and then implementing a language processor for this input language.

There are a number of advantages to taking this language centric approach to software development: (i) issues of usability of the software devolve to how well the input domain specific language has been designed; (ii) software development becomes *syntax directed*, i.e., the syntax of the DSL largely dictates how the software is developed. (iii) programs can be verified and provably correct machine code can be generated more easily. These are discussed in some more detail next.

2 Implementation Infrastructure for Domain Specific Languages

With this language based approach to software engineering, comes the problem of having to *design* the DSL, and once the language is designed, actually realizing the implementation infrastructure of the DSL. Thus, a considerable amount of infrastructure needs to be built before the first program using the DSL can be written and executed.

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 its implementation infrastructure, i.e., a program development environment (an interpreter, a compiler, debuggers, profilers, editors, etc.), to facilitate the development of programs written in this DSL. Observe that the time taken to initially design the DSL is dependent on how rapidly the DSL can be implemented, since the ability to rapidly implement the language allows its designers to quickly experiment with various language constructs and with their various possible semantics.

Note also that like any other language, a domain specific language will evolve as it is used more and more to write programs. Users and language designers gain experience with use of a DSL as more and more programs are written. This inevitably results in language features being modified or added to the DSL. If the DSL changes, then obviously its implementation infrastructure must change as well. *The success of the domain specific language methodology will depend on how rapidly this implementation infrastructure can be modified to reflect the changes in the language.* Experience has shown that the time for a domain specific language to mature can be several years [13]. A significant part of this time is spent developing and modifying the implementation infrastructure as the DSL changes. Clearly, if the implementation infrastructure can be rapidly developed/modified, then the time for the DSL to mature will be reduced, making a DSL based approach to software engineering viable and practical.

W.r.t. the dual view of a software system as a language processor of its input language, the software specification phase amounts to designing the software's input language. The actual implementation of the software system then amounts to developing the implementation infrastructure of the input language.

3 Rapid Prototyping the Implementation Infrastructure via LP

The implementation infrastructure of a DSL can be rapidly obtained by using a *Horn logical semantics* based approach [9, 11]. In the Horn logical semantics approach, both the syntax and semantics of the DSL is specified using Horn logic. Syntax is expressed using Definite Clause Grammars (DCG), while semantics is denotationally expressed as Horn clauses (the semantic specification maps software's input to its outputs). Both the syntax and semantics are executable, since they are expressed in Horn logic. Thus, the DCG specification automatically yields a parser. The semantic specification in effect is a declaratively specified

operational semantics, and together with the DCG it automatically yields an interpreter (in other words, a language processor for the DSL). Given a program written in the DSL, the interpreter can be specialized w.r.t. the program to obtain compiled code [12, 9]. Debuggers and profilers can be built by instrumenting the semantic specification [4]

W.r.t. the dual view of a software system as a processor of its input language, once the input language is designed, the software system (language processor) itself can be rapidly developed by specifying the (executable) syntax and semantics of the input language.

Note that it is not essential that one uses DCGs and declarative operational semantics for obtaining a language processor for the DSL. One could use 'C' based implementations, however, these may make take longer to develop compared to the Horn logical approach. Additionally, the other benefits of using the declarative notation of Horn clauses, e.g., being able to verify programs or automatically obtain compiled code (so that the code generation process is provably correct), may be difficult to realize. Using a denotational (compositional) approach has the additional advantage that the development of the language processor (and, thus, of the software system) becomes syntax directed. This is because in the denotational approach, there is one semantic rule per syntax (grammar) rule. Additionally, compositionality implies that the semantic rules have to be specified in a way such that the meaning of a syntactic category is given in terms of meaning of its syntactic components. Thus, the Horn logical semantics based approach provides a strong guidance on how the software is developed, making software development process faster. Thus, the main challenge that remains in the software development process is the design of its input language (i.e., its syntax), the rest quickly follows.

4 Applications

The framework described in this paper has been used for a number of applications:

1. A Domain Specific Language for solving phylogenetic inference problems in computational biology [15] has been designed; a prototype implementation is being developed.
2. A complete implementation environment for SCR [7], a domain specific language for designing real-time controllers in avionics systems, has been developed [3] including a provably correct code generator [5].
3. Software systems for translating notation from one format to another (Nemeth Braille Math code to LaTeX, computational biology formats to Nexus, etc.) have been developed [6].
4. A framework for building flexible implementations of software systems based on components has been developed [8].

Acknowledgments

I am grateful to Enrico Pontelli, Qian Wang, Hai-Feng Guo and Michael Leuschel for discussions. Thanks to Qian Wang and Ajay Bansal for help in proof-reading the paper. All errors are, of course, mine. This work has been supported by the National Science Foundation grant INT 9904063.

References

1. R. Pressman. Software Engineering: A Practitioner's Approach. 4th Edition, McGraw-Hill, 1996.
2. G. Kiczales, et al. Aspect-Oriented Programming (AOP). ACM Computing Surveys, 28A, 4 (December). 1996.
3. Qian Wang and Gopal Gupta. Provably Correct Code Generation: A Case Study. Electronic Notes in Theoretical Computer Science. Volume 118. Pages 87-109.
4. Qian Wang, G. Gupta. Rapidly Prototyping Implementation Infrastructure of Domain Specific Languages: A Semantics-based Approach. ACM Symposium on Applied Computing 2005. ACM Press. Mar 2005.
5. Qian Wang, G. Gupta, M. Leuschel. Towards Provably Correct Code Generation via Horn Logical Continuation Semantics. in Proc. International Conf. on Practical Aspects of Declarative Languages 2005. Springer Verlag. pp. 98-112. 2005.
6. G. Gupta, H-F. Guo, A. Karshmer, E. Pontelli, et al. Semantic-Based Filtering: Logic Programming's Killer App? 4th International Symposium on Practical Aspects of Declarative Languages, LNCS 2257, Springer Verlag, pp. 82-100, Jan. 2002.

7. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated Consistency Checking of Requirements Specifications. ACM Trans. Software Eng. and Methodology 5, 3, July 1996.
8. G. Gupta. A Language-Centric Approach to Software Engineering: Domain Specific Languages Meet Software Components. Proc. CologNet Workshop on Logic Programming. 2002. Madrid, Spain.
9. G. Gupta. Horn Logic Denotations and Their Applications. In *The Logic Programming Paradigm: The next 25 years*, Springer Verlag, pp. 127-160. May 1999.
10. G. Gupta. Logic Programming based Frameworks for Software Engineering. *Proc. CL2000 Workshop on Logic Programming and Software Enginering*. London, UK, July 2000.
11. G. Gupta, E. Pontelli. Specification, Implementation, and Verification of Domain Specific Languages: A Logic Programming-based Approach. In *Advances in Logic Programming: Essays in honor of 60th birthday of Robert Kowalski*. Lecture Notes in AI 2407, Springer Verlag. pp. 211-239.
12. N. Jones. Introduction to Partial Evaluation. In *ACM Computing Surveys*. 28(3):480-503, 1996.
13. 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.
14. T. L. McCluskey, J. M. Porteous, Y. Naik, C. T. Taylor, S. V A Jones. Requirements Capture Method and its use in an Air Traffic Control Application, The Journal of Software Practice and Experience, 25(1), January 1995.
15. E. Pontelli, D. Ranjan, G. Gupta, B. Milligan. Design and Implementation of a Domain Specific Language for Phylogenetic Inference. Journal of Bioinformatic and Computational Biology, 1(2):2003. pp. 201-230.