Domain-Specific Programming Languages (DSLs): a paper survey

Dan S. Reznik
CS263 Final Project
Spring 1997

1 Introduction

> Mathematicians are like the French: they take whatever you tell them and translate it into their own language – and from then on it is something entirely different – Goethe

A Domain Specific Language (DSL) is a programming language and/or environment which captures (in the form of language constructs) the vocabulray and/or abstraction specific to a particular problem domain. For example, a researcher in robotics may wish to formulate problems in motion planning with abstractions such as 'robot', 'move-arm', 'collision', 'obstacle', etc. which bring forth the salient fetures of problems commonly seen in that area as well as a simplification of the approaches to solve them. In other words, the problem vocabulary produced by a domain epert after some amount of experience tends to a formalization of the problem which is both compact and more productive. In this report I survey a number of works in DSL research discussed at lenght in the next few sections.

2 Basic Problems

Human experience is the mother of domain-specific knowledge. In particular, experience is naturally converted into a formalization of problems and a vocabulary for a particular domain. In most areas of human activity, this formalization is built within the framework of natural language, and for this reason it presents an immediate challenge to computer processing. Yet the need for the systematic transfer of knowledge from an individual's memory to “corporate” memory has obvious positive economic consequences [1]. So one of the “meta” goals of DSL research is to formalize this very process, i.e., the conversion of domain expertise into a computer-parsable language.

DSL design faces an important paradox in that it must sit between the unparsable (though ideal from an expressiveness standpoint) natural language and the criptic, CS-only lex/yacc/library-type solutions. One guiding light is the law of two R's – readability and writeability – the design must be simple enough so it is usable by non-CS-experts and it must be formal and systematized enough so it can be efficiently parsed and translated to machine executable code.

Besides accumulating a good vocabulary and placing it just at the right level of abstraction, domain experts think of problems in a particular representation – for example, electrical engineers think of circuits as two-dimensional device networks – much of a circuit's characterization is helped by reasoning *within* this spatial representation – for example, a large part of a ciruit may be ignored if it is “far” (in the euclidian sense) from an area of interest. Compare this w/ the approach taken by a circuit simulator like SPICE, where circuits are modeled as matrices and differential equations, and where the notion of physical location may not be explicitly taken into account. So a particular representation in some domain of expertise often evolves to either simplify the approach and/or to emphasize the salient features of common problems arising in that domain.

2.1 The CS way

A simplistic view for domain-specific languages is the desing of an input format. After some superficial formalization of a domain, computer scientists traditionally resort to syntax-centered tools such as lex/yacc to quickly build compilers for the newly-invented inpute language. While lex/yacc does represent an improvement over building an interpreter/compiler from scratch, these tools are stull cryptic enough that they can only be used/maintained by a programming expert. This requirement in turn often causes language design to be entagnled with the implementation, resulting in a too-CS-oriented input format. For the same reasons, a typical lex/yacc application will be forever dependent on its original designer, further complicating a flexible adaptation upon a domain-expert's request.

Another popular solution is to create a procedural library in the framework of an existing programming languge. Many examples of such libraries exist in a domain I am familiar with – Robotics; here the canonical library will contain calls to routines such as "move arm", "reset robot", etc., and manipulate objects such as "robot", "part", "obstacle", etc. Many such implementations contain an undesirable and irreversible mix of domain knowledge (built into procedures) and native-language based hacks. This is similar to the statement that a person's thoughts are somehow affected by the language they speak. So the protability and maintainability of the code is many times forever dependent on its implementor. At the domain modeling level (in this case robotics), another undesirable aspect is that disparate parts of a set of problems (involving different vocabulary) are confined to a the expressiveness and problem modeling of a single programming language – for example, while motion planning is generally thought of in spatial terms, motor control is not a spatial issue; still, the implementor is required to model both these aspects with procedural constructs.

Object orientation has been an elegant generalization of language design and a good point could be made that it is a general answer to domain-specific programming. One critique presented in [2] is that objects do not allow internal structure to be divorced from object behavior, and a program will not explicitate object interactions since these are typically encoded procedurally – for example, if a set of objects is used to represent devices in a circuit, the fact that certain devices are directly connected to others or lie in a particular neighborhood is often irrecoverable. Also, and as in the library case, object orientation does not solve the tower-of-babel syndrome in that essentially disparate pieces of a problem are forced to be modelled with a single base language. And again, such systems preserve the need for CS expertise in development, documentation, and maintenance.

2.2 Designing for the future

A basic trend in the software industry has been to tighten the engineering-customer loop [1] and in design applications so they can be maintained directly by the customer after release. It is simply too costly and unreliable to expect a typical domain expert (say a stock trader or a civil engineer) to be able to maintain either lex/yacc or a complex library of routines tailored to their domain.

In [3], *maintainability* is defined as the ease with which system can be kept in operation when modifications to the code become necessary. To this end, consider the usual modeling of problems by computer scientists as algorithms, called the *algocentric* approach [4]. Procedurally-encoded information is typically very hard to extract, so the growth and/or redevelopment of a particular application may require a type of black art called reverse-engineering.

Many domains present problems modeled after the same vocabulary and whose solutions require a similar set of techniques – for example, a financial corporation may need a program to control stock trades and another to perform corporate acquisitions, both of which require reasoning in terms of current interest rates, inflation, valuation models (often described as differential equations) – the point being that anticipating the existence of a *family* of solutions maybe crucial in cost of the design lifecycle of a (family) of software applications [1]. So in one sense, the art of DSL design is to find a reduced enough domain model (in terms of language constructs) which is general enough to accomodate for future family growth.

2.3 Cost issues

According to [1], the basic viability of a DSL is validated by the following questions: does a particular DSL solve the right problems, is it easier to use and at least as efficient as current methods, and finally, is there a significant application?

From a quantitative viewpont, [1] points out that 3-fold productivity gains can be expected from a DSL systems designed specifically for future growth. From a cost perspective, the picture is also encouraging: let us compare the costs of producing one monolithic piece of software (call it A) after the other with the design-for-growth approach in which an introductory piece of software (called B) can be easily re-adapted to accomodate for the needs of a new problem within the same domain of expertise. Let N is the number of pieces of software sold, C_a and C_b be the costs of producing A, and *modifying* B, respectively. In [1], a value of $C_b = 0.1C_a$ is suggested, though it could be easily one order of magnitude lower. Since B is designed "for the future" it will require an initial investment I, typically greater ([1] suggests three times greater) than C_a. Then for a family of N applications sold, the monolithic case would yield a total cost of NC_a while the growth-oriented project would yield a cost of $I + NC_b$. When plotted against N these two lines intersect at the "break-even" point beyond which the B solution becomes cheaper than A (in [1] it is suggested this would happen for $N = 3$). So the conclusion is rather intuitive – when the number of domain applications in the same family is likely to be 3 or greater, it does off to invest in *domain engineering* and formalize the abstractions and commonalities likely to emerge from that domain.

3 Basic Approaches

3.1 Language design 1-2-3

Pfahler and Kastens [5] present a rather simplistic solution to the design of a DSL – in their view this process should be restricted to a yes/no questionnaire: departing from a general tabula-rasa language, which more or less caputres the union of most current imperative languages (C, Fortran, Pascal, etc.), a "designer" goes through a sequence of yes/no selections which introduce contraints to suit a domain-specific need. For example, one can select whether variables are explicitly declared or not, if floating point variables are present, etc. Once this questionnaire has been filled out, the system outputs an appropriate compiler. The system contains a consistency module which ensures that user selections are consistent thoughout – for example, if a "while" statement is made part of the language, the system automatically adds the boolean type to the type repertoire. While their main claim is that language design should be quickly adaptable/modifiable, I believe their approach is a little too extreme and that it doesn't address many of the DSL desiderata such as creating an environment which is not entirely procedurally-centered and which allows problems to be modeled with a specific language and be maintained by non-experts at a non-programming level.

3.2 The FAST process

Weiss [1] outlines an architecture, called "FAST" geared toward the systematic construction of DSLs for a family of same-domain problems. While no practical implementation is presented, this work does provide good insight on the basic requirements for such a system. The author points out the need for an appropriate domain description language (or data structure) which could be then parsed for the automated finding of abstractions and commonalities within a class of problems. Such a language would have to accommodate the information appearing at each of the stages of a model for customer-centered software development, illustrated by the sequence [1]: customer, system concept, application engineering, validation, customer (iterates back), and finally to product. So an important component of the domain specification data structure are fields which will encode the information for commonalities and variabilities, the domain-specific vocabulary (say as tokens), the abstractions (which really mean the semantics of combinations of vocabulary tokens). It is hoped that if such a formalization is possible, an algorithm which translates formal specs into code becomes viable. Though similar in the documentation technique of software engineerign, one must wonder if such a domain-specification language would not suffer from a similar problem faced by DSLs – that the type of lifecycle information is typically encoded in natural language.

The FAST process also emphasizes the need to predict changes (i.e., define the future requirements of a product *family* even before it exists) to minimize software redevelopment, which, as the paper points, represents the biggest expense in the development lifecycle. With an eye on the future, programs should be designed for ease of maintenance and adaptability.

3.3 Infocentrism

In [4], Nakatani and Jones emphasize that DSL systems should make explicit the separation of declarative data ("information") from algorithms. They call this paradigm *infocentrism*. It is pointed out that the vocabulary – which they term *jargon* resulting from domain-specific experience by an individual will typically describe the problem rather than its solution (their point is that expert descriptions typically seek a vocabulary which simplifies the solution – looking a t a problem with the "right" set of galsses). To this end, DSLs should be modeled to capture information rather than procedurally-encoded solutions.

Jargons are defined much the same way as network protocols or OO interfaces, namely, as a contract between document authors and application developers. A jargon should encode conventions in a natural way which is easy to remember – the choice of terms and constraints (semantics of valid usage) should reflect domain-specific intuition – however, a specific method which allows this intuition is systematically converted into rules is not specified.

The authors conjecture that applications are likely to be more economically viable if their input format is designed and maintained by a domain expert rather than a programming language guru – similar to the new "organization of software labor" proposed in [3] which in essence coalesces jargon creator, document author, developer, and end user into a single individual. In their own words, "a domain expert designing a bad jargon better than computer expert designing good lex and yacc (but weak in domain knowledge)". While I think informal interactions between domain experts and CS experts can easily bridge this gap, I would say the strongest argument is the issue of maintenance: allowing for the domain experts to write and maintain their software and in particular, tailor them in response to growth and/or adaptation is really the key productivity issue.

An interesting point is made as to the fact that domain experts need not be non-CS people – for example, Web-applications today constitute a challenge to sane (i.e., and relatively cheap) development lifecycles in that they are a product of domain expertise in seemingly disparate CS worlds: HTML+CGL+Perl+Java(script)+SQL, not to mention graphic design, security, copyright, and marketing skills. So a point is made that important classes of problem require different domain expertise, expressed in different jargons, which must cooperate within a single monolthic framework. So a jargon-enabled language should support inter-jargon cooperation in a seamless manner.

Another important goal set forth is to design minimalist specification which supports resembles natural language as closely as possible, for reasons outlined above. Their solution is "Wiztalk", a language for domain information modeling which uses a single (and very simple) syntax across domains. A particular problem is encoded into an input "document" (containing declarative data) which is then translated into a parse tree. The algorithmic element comes in the form of "wizers" which are each associated with a particular string pattern. The interpreter binds expressions (in the input document) to algorithmic actions (wizers) whose string patterns are matched (any similarity with unix's awk is *not* a coincidence!). So mushc of the application writer's job becomes defining a vocabulary and a set of actions which are relevant to a particular domain.

The specific syntax of Wiztalk is very simple – the input document, called the ".doc" file, is made up of 0 or more expressions, each with the same syntax: semicolon followed by a term followed by a memo enclosed in square brackets. The memo itself can be treated recursively as an expression. For example, ";Buy[milk]" or ";Dan[;weight[150 pounds]]"; in the first case, the term is a verb, so its memo is the direct object; the second term is a noun so the memo is type/value information. Two remarks seem in order: first of all the ability to represent declarative information with simple syntax must be credited to a much older standard called the LISP language; secondly, the difference (other than syntactical simplicity) between a jargon and the fields of a data structure (containing both data and procedure pointer entries) is not at all clear.

With Wiztalk, the creation of a jargon does not have to undergo a lex/yacc devlopment lifecycle – the domain specialist can simply type the relevant information and then specify the semantically-relevant patterndirected actions. This allows for the seamless co-existence of several jargons within one document. However, with that, one must worry about the further readability of an input document – with so many jargons sharing the same file, documents are likely to become impenetrable, since they do not adhering to any single set of rules. In defense to the procedural modeling of a domain, in this case the causality and interference between jargons may also prove very difficult to extract. Also, in the case of a standard C library at least each module can be reverse-engineered according to the syntax rules of the C programming language.

The point is that under Wiztalk the need for jargon/action documentation is imperative. The positive side is that such documentation could be directly prepared and maintained by the domain expert, also freeing the CS expert (who implements the wizers) from the domain-specific part of the documentation. I view this as one of the best contributions of the infocentric paradigm – the work required from a CS expert is reduced to its bare bones, i.e., implementation of wizers under a well-specified set of requirements – since such specifications are passed from the domain expert (a human) to the programmer (a human) they can be done in the form of natural language, which circumvents some of the worries concerning formalization of this type of communication presented in the FAST paper above.

3.4 Aspect oriented programming

Not very different from inforcentrism, Aspect-Oriented programming (AOP) as defined by Kiczales et al. [2], represents a similar effort in dissociating domain-specific vocabulary from its implementation. As a tough philosophical issue it is pointed out that a single abstraction framework (say C, or Pascal) only does a good job of capturing one *aspect* of the program. For example, the "aspects" of a typical image processing application might consist of pixel op's, control structure, intermediate value sharing – each with its private representational and solutional idiosyncrasies. In contrast complex programs are often woven from a myriad of sources of expertise. AOP suggests that each aspect of a system's behavior should be programmed in their most natural form. The weaving together should happen last to form an executable.

The paper brings up the issue of *cross cutting*, which relates to aspects of a program getting tangled in an undocumented non-explicit fashion, both at the aspect and at the native language level. While this is undesirable, aspects will typically not be completely separate but coordinate perspectives on the total computation. So what one needs is an architecture for cooperative problem-solving in which interactions between aspects are made explicit. The solution suggested by the paper is very similar to the Wiztalk framework, in which problems are modeled into an input jargon document and then pattern matched into actions which are late-binding and maintained solely by a CS implementor. An important contribution of this paper is to define an architecture (similar to the usual AI blackboard) whereby the communication paths between aspects are made explicit as well as the data on which they concurrently operate.

4 Conclusions

A number of works in domain-specific language (DSL) research has been discussed. The basic problems in the area have been presented and a number of existing approaches described.

References

[1] D. Weiss. Creating domain-specific languages: the FAST process. In *First ACM SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997.

[2] G. Kiczales et al. Aspect oriented programming. In *First ACM SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997.

[3] A. vanDeursen and P. Klint. Little languages, little maintenance? In *First ACM SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997.

[4] L. Nakatani and M. Jones. Jargons and infocentrism. In *First ACM SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997.

[5] P. Pfahler and U. Kastens. Language design and implementation by selection. In *First ACM SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997.