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

Dan S. Reznik
CS263 Final Project
Spring 1997

> 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

1 Introduction

A Domain Specific Language (DSL) is a self-contained programming environment which captures in the form of language constructs the vocabulary and/or abstraction specific to a particular class of problems. For example, a researcher in robotics is used to describe problems with terms such as 'robot', 'joint', 'obstacle', and specific mathematical objects/algorithms such as 'convex hull', 'shortest path', etc. Likewise, a professional in finance may think of his activities in terms of 'trade deficit', 'interest rate', and certain specific mathematical models. From a programming standpoint, DSLs are expected to produce programs which are both more concise and readable than the equivalent implementation in a general PL, since DSL's constructs map directly to concepts in a given domain of expertise. 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?

One popular response to the above needs has been object-oriented programming – domain-specific concepts are mapped naturally to objects and corresponding methods. While this paradigm is general enough to express many domain-specific problems, it also requires a great deal of programming expertise, and also creates the costly need for intense communication – and in many cases eternal dependence – between a domain expert and the programming expert. Additionally, in many cases the requirements for a given software system are not well understood until late into the development process, so too much domain-specific modeling responsibility is placed on the programming expert. This in turn forces the programming expert to mix domain-specific information with the idiosyncrasies of the chosen implementation language, compromising both code re-usability and maintenance by non-experts.

The main premise of DSL research is to create a programming paradigm which very clearly separates domain expertise from programming lingo, effectively allowing a non-computer-science expert to design, develop, and maintain a significant portion of domain-specific software. It is believed that by tightening the domain-expertise/software design loop, one can achieve better productivity and appeal to a larger number of non-software experts, who driven by the forces of market, demand ever faster product turnarounds.

The remainder of this report is organized as follows: Section 2 reviews the basic motivational problems for domain-specific languages, including the existing traditional solutions and their cost implications. Section 3 reviews a number of proposed programming frameworks which address the information-separation goals of DSL research. Section 4 presents a case study of a financial DSL along with comments as to how it relates to DSL desiderata. Section 5 presents the conclusions.

2 Basic Problems

Human experience is the mother of domain-specific knowledge. In particular, experience is naturally abstracted 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-parseable language.

DSL design faces an important paradox in that it must sit between the unparseable (though ideal from an expressiveness standpoint) natural language and the available set of language implementation tools (e.g., lex, yacc, custom-tailored libraries, etc.). The principle of two R's [2] – readability and writeability – states that the design must be (i) simple enough so it is usable by non-programming domain experts and (ii) formal and systematized enough so it can be reliably and 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, and mathematicians often draw on geometry for the analysis of algebraic problems. The Amphion project [3] at Nasa leverages on these issues by making available to mathematical researchers a library of formally-documented geometrical components (routines/data structures) written in FORTRAN. Domain experts (mathematicians) state their problems using Amphion's provided components, and theorem proving is used to derive or verify certain results.

2.1 The Computer Science (CS) way

A simplistic view for domain-specific languages is the design 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 input language. While lex/yacc does represent an improvement over building an interpreter/compiler from scratch, these tools are still cryptic enough that they can only be used/maintained by a programming expert. This requirement in turn often causes language design to be entangled with the implementation, resulting in a idiosyncratic input format. For the same reasons, a typical lex/yacc application will have to depend 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 language, as in the case of Amphion above. Such implementations will invariably contain an undesirable mix of domain knowledge (built into procedures) and native-language "tricks", difficulting further maintenance by non-experts. Procedural libraries also confine disparate parts of a problem (e.g., involving different vocabulary and/or formal tools) to a single "modeling vocabulary", i.e., the constructs of the implementation language.

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 make explicit 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 not directly recoverable. Also, and as in the library case, object orientation forces different pieces of a problem to be modeled with a single base language. This is in turn compounded to dependencies on CS-expertise in development, documentation, and maintenance.

2.2 Maintainable software

A basic trend in the software industry has been to tighten the engineering-customer loop [1] in designing 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 [4], *maintainability* is defined as the ease with which a 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 [5]. It was mentioned in the previous section that procedurally-encoded information is typically very hard to extract, so in many cases the growth and/or redevelopment of a legacy application will involve much time-consuming, unsystematic reverse-engineering. One proposed solution has been to shift the focus of the domain-modeling phase to declaring *what* is known about a problem, rather than *how* a problem is to be solved. This is called the *infocentric* approach after [5]. One advantage of this method is that it lets the domain expert focus on what it knows best – the description of a problem in terms of an optimal vocabulary.

2.3 Identifying families

Maintainability (and development turnaround) can also be improved by noticing that 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]. In fact, one key to DSL design is to find a reduced enough domain model (in terms of language constructs) which is general enough to accommodate for future family growth.

From a quantitative viewpoint, [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 accommodate for the needs of a new problem within the same domain of expertise. Let C_a and C_b be the costs of producing A, and *modifying* B, respectively. In [1], a conservative value of $C_b = 0.1C_a$ is suggested, though it could easily be one order of magnitude lower. Since B is designed "for the future" it will require an initial investment I, typically greater than C_a ([1] suggests three times greater). 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; with the above constants this would happen for $N = 3$. The conclusion is rather intuitive – when the number of domain applications in the same family is likely to be 3 or greater, it pays off to invest in *domain engineering* and formalize the abstractions and commonalities likely to emerge from that domain. Of course one of the main challenges is how to recognize this situation and systematize the development process.

3 Basic Approaches

After having reviewed some of the goals and issues of DSL, a few existing approaches are presented. They range from very simplistic approaches to frameworks for completely automated language generation. Unfortunately, working implementations tend to materialize more often at the simplistic end of this spectrum.

3.1 Language design 1-2-3

Pfahler and Kastens [6] 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 captures the union of most current imperative languages (C, Fortran, Pascal, etc.), a "designer" goes through a sequence of yes/no selections which introduce constraints 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 throughout – 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. An important component of their *domain specification data structure* are fields which will encode the information for commonalities and variabilities, domain-specific vocabulary (say as tokens), and abstractions (which they oversimplify to 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, that being the ultimate goal of DSL research.

A strong component of the FAST process is that it 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 they claim to be the biggest expense in the development lifecycle. They are probably the first to provide an implementable framework (essentially a data structure 'generalizer') which addresses this problem.

3.3 Infocentrism

In [5], 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 – they call it a *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, i.e., are *goal oriented*. To this end, DSLs should be modeled to capture information rather than procedurally-encoded solutions.

Jargons are defined in 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 to be systematically converted into rules is not provided.

The authors subscribe to the previously-mentioned notion 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 [4]. In essence, this coalesces jargon creator, document author, developer, and end user into a single individual. They claim that "a domain expert designing a bad jargon is better than a computer expert designing good lex and yacc (but weak in domain knowledge)".

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 worlds (all of which are CS worlds!): HTML+CGI+Perl+Java(script)+SQL, not to mention graphic design, security, copyright, and marketing skills. A point is made that certain problems require different domain expertise (even if within the same professional niche), expressed in different jargons, which must cooperate within a single monolithic framework. So a jargon-enabled language should support inter-jargon cooperation in a seamless manner.

Another important goal set forth is to design a minimalist specification which supports and 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. An interpreter binds expressions in the input document to algorithmic actions (which they call *wizers*) by simple pattern pattern-matching. They admit this design to be inspired in the *awk* Unix tool. Under this framework, much 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.

A relevant comment is that the ability to represent declarative information with simple syntax must be credited to a much older standard called the LISP language; in fact, 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, except that a jargon's fields are in essence random and unlabeled. But in Wiztalk's defense, the creation of a jargon does not have to undergo a lex/yacc development lifecycle – the domain specialist can simply type the relevant information and then specify the semantically-relevant pattern-directed actions. This allows for the seamless coexistence 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 adhere to any single set of rules. Also, and in parallel with procedural modeling, Wiztalk will suffer from non-explicit interferences between jargons, making a program difficult to maintain (an/or understand). From a reverse-engineering standpoint, a jargon file has an added problem of non-adherence to any strict syntactical scheme; compare this with at least the causality and well-documented execution order of most imperative programming languages (one can always trace the 'main()' function in C).

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, completely designed by the domain experts – 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 infocentrism, Aspect-Oriented programming (AOP) as defined by Kiczales et al. [2], represents a similar effort in dissociating domain-specific vocabulary from its implementation. An important issue pointed out is that a single abstraction framework (say C, or Pascal library) 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 (thus addressing the main problem with jargon files). The framework suggested by the paper is similar to Wiztalk (above), 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. However, AOP also defines an architecture which requires both the communication paths between aspects and the resources they manipulate (such as memory or peripherals) to be explicitly declared as part of the program.

3.5 DSL's desiderata

From the above works, a list of the principal desiderata of DSL can be inferred. Reasearch in DSL is about:

1. Defining a language which allows for the easy expression of domain-specific jargons.
2. The language should be analyzable for the extraction of vocabulary similarities across applications (family issues)
3. Encouraging domain-specific knowledge to be expressed declaratively rather than algorithmically
4. Explicitating paths and resources between jargons
5. Allowing jargons to co-exist
6. Designing with the awareness of cost issues (monolithic vs. maintainable software)
7. Relieveing the CS-expert completely from domain-expertise research.

4 The Case Study of a Financial DSL

Here we present some details of a DSL currently being used by at least one major bank in the Netherlands. The language is called Risla (Dutch acronym for Interest Rate Information Language), documented partially in works [7] and [8].

Risla is a DSL for the manipulation of *interest rate products*, which consist of financial instruments such as loans, futures, forward rate agreements, etc. All these instruments possess by nature much common vocabulary and concepts. Financial instruments typically involve involve cash flows (time series of input/output of capital) and the assessment of worth based on multiple interest rate scenarios.

The financial industry has also been strongly influenced by 3 decades of "business programming", mostly (if not entirely) in the COBOL language. For years, the particular Dutch bank which commissioned Risla's development, had relied on a large library of well documented COBOL procedures, which Risla was expected to take advantage of.

Risla's goals were set to (i) reduce the time-to-market of new financial products (constantly being "invented" by any given bank), while (ii) leverage on the existing COBOL libraries, (iii) allow for end-user (domain expert) maintenance and programming, and (iv) make applications easily adaptable or upgradable to closely-related applications.

The most important parts of a typical Risla program are an ordered series of cash flows – (amount,date) pairs – and rules defining how to compute cash flows from initial contract values. A compiler is provided which translates Risla programs into COBOL programs making use of the existing libraries, but this is done entirely transparently to the end-user. Essentially, Risla defines a class (in OO terms) whose fields can be filled-out by finance experts through a text interface. This process will often leave functionality "holes" behind which must be filled by programming experts.

An example of a simple Risla program, describing a "loan" product is shown below. Comments following some lines are preceded by the %% characters. Lines are shown numbered. The program's syntax resembles Pascal's program/var/begin-end familiar structure. It begins with a title (LOAN), followed by a declaration, followed by an implementation part. The declaration defines the fields of the "contract" data structure, each with an appropriate type. The information part defines fields which are inspectable from the outside. The registration fields identify the set of routines which allow data to be entered into a contract (in this case, a redemption). Computations such as FPA and FRDM are defined which will be used as queries by financial analysts to evaluate the merit of the given financial contract. On the implementation part of this module, such computations are linked to procedures available from the legacy COBOL financial library. Risla also allows for error/exception messages present in the code below.

```
product LOAN

declaration
 contract data
    PAMOUNT : amount                 %% principal
    STARTDATE : date                 %% start
    MATURDATE : date                 %% maturity
    INTRATE : int-rate               %% Interest rate
    RDMLIST := [] : cashflow-list    %% list of redemptions
                                     %% initially empty
 information                         %% info that can be extracted
```

```
   PAD : cashflow-list              %% principal cashflow
   IAD : cashflow-list              %% interest cashflow
registration
   RDM(AMOUNT : amount,
       DATUM : date)                %% register one redemption
local                               %% computable cashflows
   FPA(CHFLLIST : cashflow-list) : amount   %% final principal amount
   FRDM : cashflow                  %% final redemption
error checks
   "Wrong term dates" in case of STARTDATE >= MATURDATE
   "Negative amount"  in case of PAMOUNT < 0.0
implementation                      %% how to compute cashflows
local                               %% define in terms of built-in
                                    %% cobol functions
   define FPA as IBD(CHFLLIST, PAMOUNT, MATURDATE)
   define FRDM as <FPA(RDMLIST), MATURDATE>
                                    %% IBD=initial balance on date
                                    %% CIA=calculate interest amount
information
   define PAF as [<PAMOUNT, STARTDATE>] >> RDMLIST >> [FRDM]
   define IAF as [<CIA( BL(RDMLIST, <PAMOUNT,
                    <STARTDATE, MATURDATE>>), INTRATE ), MATURDATE >]
registration
   define RDM as
      error checks
         "Date not in interval" in case of (DATUM < STARTDATE) or
                                           (DATUM >= MATURDATE)
         "Negative amount" in case of AMOUNT <= 0.0
         "Amount too big" in case of FPA(RDMLIST >> [<AMOUNT, DATUM>])
                                    > 0.0
      RDMLIST := RDMLIST >> [<AMOUNT, DATUM>]
```

The authors claim that the problem of migration from an existing product to a similar one, has been addressed by allowing for the parametrization of existing objects (a form of generalization), similar to type parametrization in C++. Overall, they claim that with Risla product-to-market time has been reduced from 3 months to 2-3 weeks.

Whether the above is true, one must be somewhat disappointed with both the similarity of Risla with existing general-purpose PLs and also with the re-phrasing of well-defined OO concepts which are also standard in modern PLs such as C++. Risla's program is, in many ways, the definition of a C++ class with Pascal pseudo-syntax. Indeed, it is editable by non-domain experts but only probably after much interaction with the computer experts. One if left to wonder if for the proper exploitation of Risla, a finance expert is not forced into an exercise in general-purpose programming, in which case, he or she would be better off learning a general-purpose language anyway.

5 Conclusions

A number of works in domain-specific language (DSL) research has been discussed. The fundamental goals in the area are (i) end-user "empowerment" (for the purposes of development and maintainability), and (ii) the clear separation between domain and computer expertise in large programming efforts. A number of existing approaches and some of the more important issues in DSL research have been described. One commercial DSL targeted to the financial domain has been explored, however the implementation looks at best naive and subsumed by existing general-purpose PLs.

As a final comment, one threat anticipated for DSLs is the increasing popularity of languages for "mass-programming" such as Visual Basic, which possess as advantages their low price, general purposeness, visually-familiar programming tools, and a large user base (more testing and less bugs). A parallel phenomenon is happening in the digital signal processor market – the concept of a dedicated chip for signal processing is slowly giving way to general-purpose microprocessors which appeal (due to low prices and comparable performance) to a much larger audience. Their non-signal-processing-specific design is compensated by the availability of abstraction tools that perform domain-specific tasks seamlessly. The forces of adhering to a standard (e.g., using Microsoft programming tools rather than some obscure DSL) may drive DSLs out of business in a similar way.

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] M. Lowry, A. Philpot, and I. Underwood. A formal approach to domain oriented software-design environments. In *KBSE'94*, 1994.

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

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

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

[7] A. van Deursen. Domain-specific languages versus object-oriented frameworks: A financial engineering case study. In *Proceedings Smalltalk and Java in Industry and Academia*, Erfurt, Netherlands, September 1997.

[8] B. Arnold abd A. van Deursen and M. Res. An algebraic specification of a language for describing financial products. In *Proceedings of the ICSE-17 Workshop on Formal Methods Applications in Software Engineering Practice*, Seattle, WA, April 1995.