Lessons Learned from Real DSL Experiments

David Wile
Teknowledge Corp.
dwile@teknowledge.com

Abstract

Over the years, our group, led by Bob Balzer, designed and implemented three domain-specific languages for use by outside people in real situations. The first language described the communication format of messages used by NATO to specify command-and-control messages between people and equipment; the processor we generated checked these messages for consistency. The second language was in part graphical, part textual, and was used to demonstrate how naval ship formations were constituted and the constrained movements they could undergo. The last language was a mixture of graphics, text, and declarative information specified using three different COTS products. It was used to describe census survey "instruments," used to collect census data in the field. The code generated was to be installed in the survey takers' laptops.

Each of these was actually a prototype for what would have taken more time to engineer and polish before putting into practice. Although each effort was essentially successful, none of the languages was ever followed up with the subsequent engineering efforts that we expected or at least hoped for. The first two were demonstrated and reviewed informally. The last effort was more seriously reviewed, in that training sessions and a formal review process were undertaken to evaluate the potential effectiveness of the product. Herein I elaborate where these language efforts succeeded and where they failed, gleaning lessons for others who take the somewhat risky step of committing to develop a DSL for a particular user community.

1. Introduction

We have undertaken three DSL experiments over the last 10 years. The nature of the languages we designed for each progressed from purely syntactic designs, to mixed graphics and syntax designs, and finally to a mixture of graphics, syntax, database schema, and web forms designs. Purists may not want to call the second and third experiments "languages," but the inclusion of these as languages has to do with how our notions of what constitutes domain-specific language support for problem domain experts have evolved

from the early days. It is clear that graphical representations are often superior for expressing relationships that are somewhat difficult to extract from a textual presentation. Hence, we have come to include both graphical and textual modes of expression as language design activities. Moreover, there are many other activities that are entirely analogous to language design, such as database schema design, UML model design, even data structure design, and abstract syntax design, such as DTD / XML. Some of these can be used to construct a "poor man's DSL" [16]. The lessons we learned presented below can usually be applied to these activities as well as to the purely syntactic or graphical DSLs.

As the experiments below are sketched, the lessons learned will be interspersed with the description. The reader should not read these lessons as absolutes. In essence, the lessons simply identify important issues of which one should be aware. After the lessons have been introduced, a synthesis will be presented that should help to organize the approach to DSL technology design, development and adoption.

2. Military Message Experiment

Our group had been involved in specification language design since the mid-1970s [5], but in the mid-1980s we noticed that our specification languages were still quite cumbersome in any particular problem domain. Around then we described the idea of "local formalisms" [12], little languages that covered exactly what you wanted to say and no more. Fortunately, the syntax-driven tool support we had built for general purpose program language design, analysis, and compilation was readily applicable to such DSLs as well [13].

Our first local formalism that was not designed for the computer professional was a language for describing the format of messages that conformed to a NATO communication standard [3]. This standard describes legal messages transmitted between all kinds of vehicles and processing stations, from tanks, to planes, to command centers. This format had never been described formally in any way other than through the millions of lines of Ada code written to check conformance to the format, admittedly for a

large variety of types of messages.[1] Messages between various military and civilian organizations and devices use this format - even now, I suspect -, probably invented back when the only reliable electronic recording medium was punched paper tape!

The message structure was very simple. A message comprised a sequence of so-called data-sets made up of lines of ASCII. Each line began with an operator, was followed by a sequence of fields separated by a "/" character, and was terminated with "//". Figure 1 illustrates a simple snapshot from such a message.

important this is.) One day, after trying one design for specifying the MTSL our advocate explained that yes, he could easily express everything he saw using it. He showed us a rather formal document describing the allowed fields and line sequence for each data-set type that he got from NATO. He was correlating our new language specifications with those in the document, rather than simply using the document as the specification! Figure 2 shows a snapshot of such a specification.

Oddly enough, this language had never been formalized and used in tools that processed the messages, but its usage

---

GRAPH /PLAN/TRIANG/250833Z/AREA GREEN/15SUP398644/15SUP439640/15SUP403622//
Figure 1: Message Dataset Example

The format of the various fields, whether they were required or optional, and whether they could occur multiply, was described separately for each operator and re-used for all of the various types of messages; we called this (portion of) the language, the Data Set Specification Language, or DSSL. The allowed set of operators, the sequences in which they could occur, and their multiplicity were described for each type of message; this language was called the Message Type Sequence Language, or MTSL. There were hundreds of data - set operators and scores of types of messages. Within each sub-language there were further constraints on the various fields and non-syntactic relationships between the lines that had to be maintained.

We designed a language to describe these validation constraints along with a program generator of Ada code to run the validations. We also designed a language to describe how to update a database with the information each message contained; our program generated SQL code to perform the updates.

We learned our first lesson rather quickly:

was clearly very strict and represented 100s of pages of specification, already done and debugged! We adopted it immediately in place of a part of the MTSL that we had designed, even though this notation was horribly designed from a programming language design point of view. Notice that the open parentheses at the beginning of a line - indicating the level of nesting of logical groups of datasets - were never closed!

The three separable languages, the DSSL, MTSL, and Updates languages were designed to maximize reuse of constructs in different contexts. Operators on grammars [14] were used to form the three languages, including a way to union grammars and to instantiate parameterized grammars. Since all three of these languages are used by different people in the organizations defining the specifications of messages, datasets, and DB updates, this normally good language design principle, of reusing like constructs for similar activities, was largely wasted in this situation!

Lesson 2: Understand the organizational roles of the people who will be using your language.

Lesson 2 Corollary: Understand the present solution design

```
type OWN SITREP =
        (0)    REFTIME    /*M/M/7              REFERENCE TIME
(       (M)    ORGID      /O/M/H/O/M/M/O//    ORGANIZATION DESIGNATOR
(       (0)    ORGSTAT    /O/O/O/O/O/O/O//    ORGANIZATION STATUS
(       (0)    TIME       /*M/M//             TIME
((      (0)    LOCATION   /M/M/O/O/O/O/+O/7   LOCATION
(( *    (0)    GRAPH      /M/M/O/O/*O//       GRAPHIC DESCRIPTION
((      (0)    DRCTN      /M/O/O//            DIRECTION
```
Figure 2: MTSL Type Specification Example

Lesson 1: Adopt whatever formal notation the domain experts already have, rather than invent a new one.

We had an advocate for our (proposed) DSL technology in close contact with sympathetic experts in the domain. (Not until the third experiment did we realize how crucially

process thoroughly before undertaking to substitute a DSL approach.

The DSSL is really only used by a very few people designing NATO-wide standards. The MTSL is used in more specific domains. And the update activities vary by command center. Our initial misconception that the DSSL and the MTSL would be used together was suggested by our experience with program language design, where a set of declarations is used to shape the use of other constructs. For

---

[1] We later discovered that an OGI group, led by Kieburtz and Hook, was actually implementing an analyzer for the same domain, using abstract interpretations in Haskell [7].

example, an array declaration might set up the use of bracketed subscripts in the program proper. Here the people doing the declarations may have done it years before the message types using them are defined. On the other hand, the reuse made our language design job somewhat simpler. In fact, the design of these languages illustrates another DSL design heuristic:

Lesson 3: You are almost never designing a programming language.

Lesson 3 Corollary: Design only what is necessary. Learn to recognize your tendency to over-design.

Most DSL designers come from language design backgrounds. There the admirable principles of orthogonality and economy of form are not necessarily well-applied to DSL design. Especially in catering to the pre-existing jargon and notations of the domain, one must be careful not to embellish or over-generalize the language. It is worth pointing out that in the DSSL, MTSL and Updates languages, there is no notion of sequentiality; moreover, all conditionality is bound up in the logic operators of the predicates. Because there is a notion of repeated dataset usage in some message types, the notion of iteration was introduced in all three languages. However, it was specialized to only allow iteration over the fields in the message or datasets in the message type. No extraneous computational power was added.

It will probably be unusual to be able to cover all situations with a simple language that expresses the nuances of all activity in the problem domain. Here, there were constraints on some fields that simply would not yield to expression in logic or via restrictions on alphabetic or numeric content, e.g., valid dates. Hence, a facility was provided for the user to refer to externally defined predicates, support for which was beyond the expressiveness of the language.

Lesson 4: Strive for an 80% solution.

If a large percentage (e.g. 80%) of the activities can be compressed using the DSL paradigm, the experts have plenty of time left over to deal with the hard or interesting parts [8]. For example, here programming experts in the domain can provide the definitions for the predicates such as valid dates. Another place this lesson has an impact is on the architectural framework that specifications in the DSL compile into or otherwise interact with. A distinct advantage of a domain-specific approach is that simplifications of the language can arise from implicit assumptions one can place on the evaluation environment. Here, for example, we were able to "type check" the SQL statements against separately-specified, pre-declared database schema, thus guaranteeing a lack of run-time errors due to schema mismatch. We needed no separate schema declaration language within the DSL itself.

Lesson 5: Leverage the infrastructure you are providing to reduce the language complexity.

In the end, we were somewhat surprised to experience a 50 to 1 improvement in lines of generated code vs. lines of specification [1]!

Lesson 6: Strive for, and expect, large improvements in expert productivity!

3. Replenishment at Sea

The second experiment we undertook was to specify naval ship formations and the movements of ships within the formations when doing particular maneuvers. A particular example maneuver was specified. It was somewhat complex, in that a single supply ship, filled with both fuel and supplies, was used to refuel and resupply each ship in a convoy of ships accompanying an aircraft carrier. Helicopters ferried goods from the supply ship to the surrounding ships during the fueling maneuver. Fueling occurred one ship at a time, but alternate sides of the supply ship were used to speed up matters. During the refueling, the convoy surrounded the fuel ship in a protective shield. There were myriad details involving what happens to the aircraft carrier after fueling, how the ships moved into formation as each new ship pulled along-side the supply ship, and how the "guard ship" - so

Replenishment At Sea - Movement
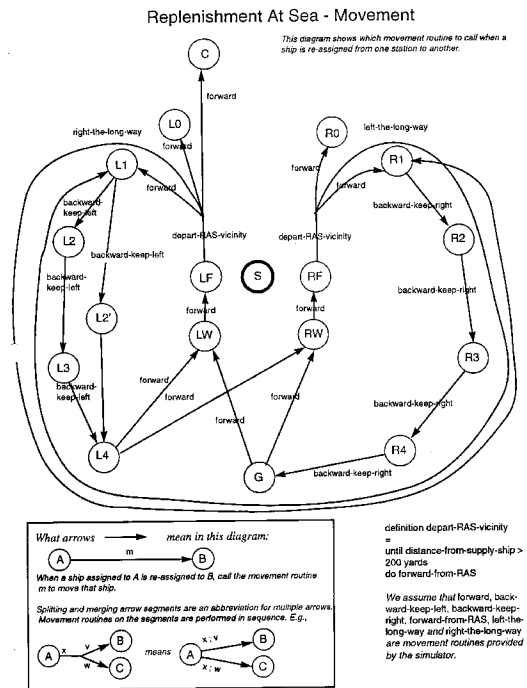
named for its role in guarding for man-overboard situations - was to move.

A lesson we learned immediately here was the importance

of graphics for some specification languages.[2] Experts in the domain sketched the charts for maneuvers routinely.

Lesson 1 Corollary: One should look to informal notations of the domain as the foundation for the DSL.

Moreover, several graphical views were useful for different purposes, such as specifying the initial layout of ships, the movement patterns, the specific movements in particular maneuvers, etc. In general:

Lesson 7: A mixture of specification techniques may be necessary to facilitate expression of the appropriate domain expertise.

Figure 3 - presented here for its "gestalt" nature rather than to be understood out of context - illustrates a view of information used in the replenishment maneuver specification. Views such as this were augmented with declarative information about how maneuver steps were executed as expressed in Figure 4. Although we had the appropriate technology for translating this language [13], it was actually translated manually to an internal representation based on relationships in a language called AP5 [4]. The infrastructure into which the RAS specifications were translated was known as MODSAF - Modular Simulated Automated Forces, a networked system of simulation modules used by the military to simulate military maneuvers among several participants. There were already display mechanisms in place to show how the various participants were moving, so visualization problems were solved for us by the infrastructure that we assumed to be present.

```
action definitions:
  definition hookup[ship] = wait 20 minutes
   i.e., simulate hookup of a ship by waiting 20 minutes.
  definition fuel[ship] =
     wait fuel-rate(ship) * fuel-room(ship);
     assert fueled?(ship)
  definition breakdown[ship] =
     wait 15 minutes; assert brokendown?(ship)
  definition emergency-breakdown[ship] =
     wait 5 minutes; assert brokendown?(ship)

  predicate definitions:
   definition left-side-has-room? =
     not (every left-side-stations, s, is asg?s)
   i.e., true if not every left side station
       has a ship assigned to it. Similarly,
   definition right-side-has-room? =
     not (every right-side-stations, s, is asg?s)
```

Figure 4: Action and Predicate Definitions

---

[2] We never produced the tools to automatically process the graphical input; instead, we manually translated to an internal relational formalism to represent the graphical content. A follow-on project would necessarily have had to process the graphics, however, for the system to be used by the domain experts.

However, we did not have a strong domain-expert advocate for this program and had to invent the algorithms for turning the ships, timing the events, etc. These were somewhat difficult for us and were certainly beyond our expertise for producing a polished product. As a demonstration, the algorithms worked well enough. We had difficulties adapting the MODSAF framework for the purposes of ship movement simulation, in that some assumptions about artillery - the original domain for which the framework was designed - were built into the framework! For example, we could not understand why our ships could not be placed closer than 100 feet from one another. It turned out that the simulator took the ship's length and rotated it about its center to determine the object's "boundary"!

Lesson 8: Establish close ties with a domain expert to produce the infrastructure that the system will be translated into.

The RAS system never was critically reviewed for further development. Rather, a demonstration was given, the experiment was declared a success, and the thread was dropped from lack of a strong advocate within the customer's organization.

4. Survey Instrument Creator (SIC)

The third experiment was to develop a system for designing and creating so-called "survey instruments" for census takers, the software that runs in the laptops they take with them to interview people during census taking[15]. The instruments themselves behave in a well-defined manner, where a series of questions is presented to the census taker, who elicits answers from the interviewee and enters them. Subsequent questions are chosen, in part, based on answers given. For example, if the residence is established as a "farm dwelling," there will likely ensue a variety of questions relevant to farming, such as equipment owned, other buildings on the property, whether people live in them, etc. These instruments are quite complex, comprising potentially thousands of questions; only a relatively small subset will be visited in any particular interview. There are also complicating factors such as unknown answers, interrupted interview resumption, and inconsistent responses that cause the interview to revert to a previous point, etc. Naturally, each of the answers is entered into a database in the interviewer's laptop to be taken back and integrated into the national census. (Actually, different kinds of censuses are taken for different purposes, and the system we were designing was to be useful in the design of any of these.)

We decided to make the instrument appear to simply give the interviewer a sequence of relevant html pages in a browser on the laptop. The sequence chosen was to be driven by a flow-chart language used by the instrument designer. A local database was used to accumulate the
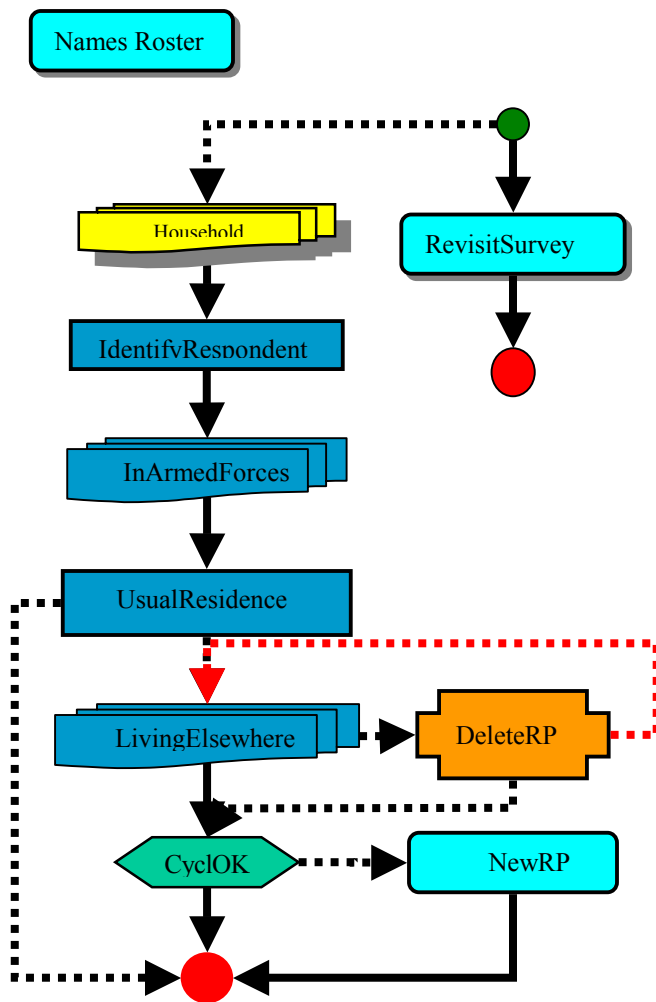
responses extracted from each browser page.

Figure 5: A Census Survey Flow-Chart Specification

Having had the experience of designing the other domain-specific languages, we intended to: use the jargon of the domain experts, provide design tools that were not overly general, use graphical representations where appropriate, and to provide as many views of the information required as were relevant. We had also become adept at adapting COTS tools for purposes unintended by their designers [2]. In particular, our PowerPoint Design Editor was developed as an extension to PowerPoint that aided with the definition and use of graphical styles as direct analogues to domain-specific language designs [6, 16].

To specify and analyze the census surveys, we integrated three COTS tools - our PowerPoint-based Design Editor, the Microsoft Access database toolset, and a lesser-known web page development product, known as Tarantula. The DesignEditor was used to describe the interview flow, in a flow-chart-like language specialized to the particular domain. The Access database described the structure of the data to be

collected. The Tarantula form-designer was used to provide the detailed questions that were asked and the appropriate layout for optimal data collection. Moreover, we used conventional syntax-directed language processing techniques for some of the textual attributes associated with the graphical entities.

Figure 5 shows an example portion of a flow-chart specification. It is simply a PowerPoint slide, constructed using symbols from a pallet generated from a graphical grammar delineating what will be understood by analyzers of this domain-specific language [6]. This slide defines the Names Roster process. The start and end symbols - circles of green and red in the original presentation - define the entry and exit to a subprocess. A subprocess is invoked using the icons with the rounded ends, RevisitSurvey and NewRP, for examples. Overlapped image icons represent the fact that a set of answers will be collected, for example, in InArmedForces. The various connection types have distinct meanings as well. The dotted ones are labeled with conditions under which a transfer is to be made, the solids being otherwise branches. The return connection from DeleteRP to LivingElsewhere is actually a different kind of connector and is the only kind allowed to loop back in the question-asking process. This represents discovery of an inconsistency and the database must be rolled back.

Attached to the icons are properties that further describe the information to be collected, how to put it into the database, and what forms to use to obtain the answers from the interviewees.

An SQL extension was used to express the database activity, while the conditions on connectors were described with simple predicates or elements of enumerated types. Analyzers were developed that checked for inconsistencies in the flow-chart flow, for use of questions that involved undefined Access database fields, for database fields which there were no questions asked to fill, and for branch conditions left uncovered by the instrument. We intended to develop a generator of the program to be run on the laptops to actually take the surveys, but this was put off to the second phase of product development.

The web form attribute to use during the interview was automatically filled in when the Tarantula tool was visited with the corresponding flow-graph box "selected" in PowerPoint. The form designer used a simple set of icons to construct the questionnaire. A specialized language was designed to allow the form designer to cause context-sensitive text to appear in the questions. For example, one would not want to refer to "his or her" age when you already know the gender of the person you are discussing! Hence, one could condition the questions to be asked by access to the database to the gender of the person. The forms were also set up to allow entry of tabular data, filling in attributes of several so-called "roster entries" in one session. Providing

these abilities was expected of us by the domain experts and not something we imposed on them as new.

A sample form is presented in Figure 6. The Instructions, Probe and Help frames are available for every form, allowing the designer to give the interviewer general instructions, a hint for how to probe more information from the interviewee and the potential for additional help if the interviewee is having trouble. In this form, the prompt is computed based on how many members of the household are present. Hence, the question would read "Are either of you now on full-time duty in the active Armed Forces" if there were two household members. The form will actually display a table of persons - their numbers and their names - along with a check box to
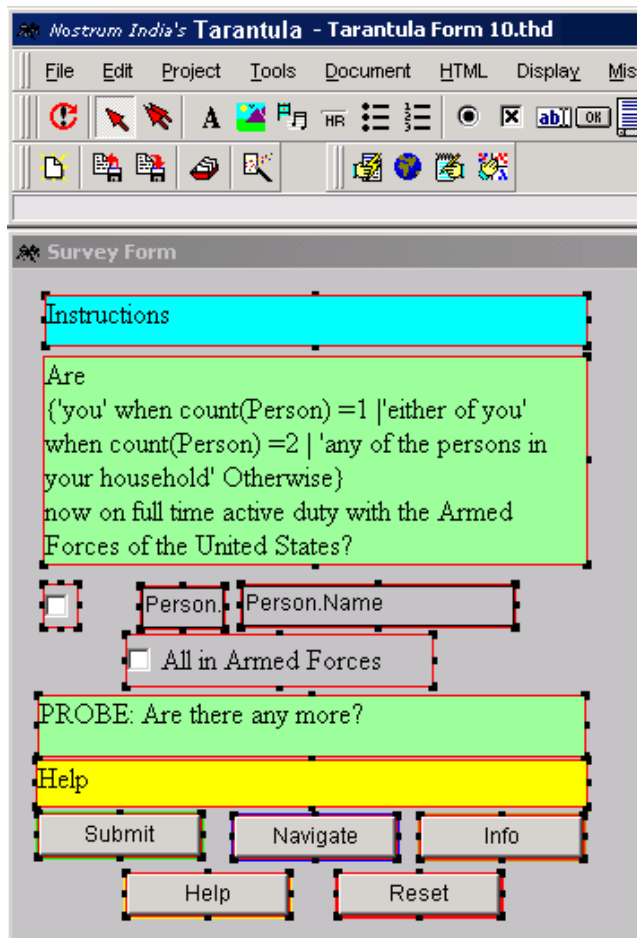

Figure 6: A Sample Tarantula Form

indicate that they are armed forces members. Notice the ability to check a single box that all are so employed.

While one tended to alternate between describing forms and describing the interview flow, describing the Access relations used to hold the data collected was done as a parallel activity, as is shown in Figure 7.

The attributes of each roster element were described in a separate relation, such as person, dwelling, family

Figure 7: Access Definition of Household Rosters

relationship, etc. The example shows (parts of) the Household relation, which established the context for the particular interview being described. Along with actual attributes of the house itself, such as number of bedrooms, kitchens, etc., were attributes specific to the interview, such as the case identifier, the REFPER (reference person, generally the owner of the house) and the RESP, (respondent being questioned). The designer of the database could design enumerated types and limit the responses to those types. These were in turn checked for coverage in the flow diagram when a question discriminated based on an element described as one of the enumerated type.

5. Evaluation

The first two experiments were done in the early- and mid- 1990s, respectively; they were never formally reviewed. The NATO message experiment was followed up by a serious attempt to adapt the system to an existing Ada-based system for message error diagnosis and repair, but the funding ultimately ran out before a final product was produced. Similarly, the RAS experiment was followed up by an experiment describing tank movements and formations, modeled again in MODSAF, but it ultimately faded out as funding diminished.

The Survey Instrument Creator project was structured rather formally. It was funded in part by the United States Census Bureau and by the National Science Foundation. It evolved into a two-part project. A demonstration system (described above) was developed, documented, and 12 people were trained in its use. This workgroup comprised evaluators from: the United States Census Bureau (5), the National Center for Health Statistics (NCHS) (3), the Bureau of Labor Statistics (BLS) (1), the Bureau of Justice Statistics (BJS) (1), and the Energy Information Administration (EIA) (2). Their average experience was around 15 years each! They subsequently evaluated the results [11], with the intent to decide whether to provide more funding to develop a production system to be used in the field by census takers.

The evaluation was organized about 5 key objectives that

the project purported to achieve:

1. Flowcharts help the user design the instrument.
2. The tool is intuitive because it relies on COTS (Commercial, Off-the-Shelf) products where users already have familiarity.
3. The integration of the three components of survey design - Metadata Design, Administration Design, and Form Design - in one software system helps streamline the process of instrument development.
4. The survey instrument can be automatically generated, thus eliminating, or greatly reducing, the role of the computer programmer or instrument "author" in creating the survey instrument from detailed specifications.
5. The use of COTS software to develop SIC is a good approach.

The results of the evaluation were mixed. In retrospect, we could have predicted some of them had we been more careful to pay attention to the lessons we had already learned in the other experiments. Below we consider the positive aspects first, followed by the negative aspects, interspersed with new lessons and admissions of which lessons we ignored.

The good news is that the evaluators liked the overall idea of describing the flow of the interviews in a higher level notation than using programming notions or even paper-based methods with explicit instructions on where to look next in the document. To quote the evaluators:

> The general concept of using flowcharts to help design survey questionnaires and instruments was liked by a majority of the evaluators. ... Specifically, the evaluators liked the:
> - Graphical objects chosen for the flowcharts
> - Ease with which it was possible to create flowcharts
> - Ability to have embedded models (flowchart within a flowchart)
> - Preview function used to view the flowcharts
> - Use of flowcharting software to develop detailed specifications
> - Use of flowcharts as a diagnostic tool to examine the structure of survey instruments and test the completeness of paths between survey items
> - Use of flowcharts to provide a modular view of the questionnaire
> - Use of flowcharts to document the questionnaire

That being said, they had many concerns that will be touched on in the negative aspects below. The second positive response was to the third objective, of mixing the development of database (metadata) schema, interview flow and questionnaire development. Again, to quote the evaluators:

> The Evaluation Workgroup liked the concept of tying together the various aspects of survey design - specifications, forms design, graphical flow, metadata, data storage, and administration of a survey instrument, into one software system. In particular, defining metadata

at an early stage would allow reusability of data definitions throughout the survey lifecycle. ...

The other three objectives were less successful according to the committee members. They had several concerns about the role of COTS as the user interfaces to their tools. They found them to be unintuitive and to require considerable training. They apparently do not design survey instruments as a routine activity, but rather as something that occurs sporadically, for one complaint was that: "the tool [would not] be easy to learn and easy to remember how to use after long periods of inactivity." We would certainly never claim that PowerPoint or Access had either of these properties (and Tarantula was admittedly just a "place-holder" for a more widely accepted form-design tool). Our hope would be that they would use the COTS tools in other activities in their daily lives, e.g., as many military employees do. We clearly failed to:

Lesson 9: Understand the background expertise of the people affected by the DSL technology introduction.

Another problem with COTS concerned the maintenance of tools as the COTS tools evolve, as they are sure to do. In fact, this concern had to do with the very structure of their tool support system, and is something we overlooked from the beginning. To quote "Most federal statistical agencies do not want to get into the CAI maintenance or development business." There was probably some confusion on their part here, but this is a serious concern in general:

Lesson 10: Be sure that the intended technology transfer process from your product into their organization's infrastructure is consistent with their business model.

This lesson covers a multitude of potential problems with inserting the technology into the organization; these solutions can be very difficult.

But actually, some of the lessons learned were simply reinforcements of what we knew, but failed to practice. For example, we insisted on using the term "metadata data item" for what they referred to as a "variable." This certainly violates the obvious lesson:

Corollary to Lesson 1: Use their jargon terms whenever possible.

To quote the evaluation report further:

> First, it is important to use common terms familiar to the users, rather than terms familiar to the software developers. ... The software developer should adapt to the user, rather than the other way around, for items that are not significant. It helps the user understand new software if they can relate concepts to things they already know. Also, it helps the user maintain a positive attitude toward the software and the developer's ability to design software for the user's environment.

Other lessons were somewhat surprising, for example, our inadequacy in the role of teacher and trainer (neither Bob nor myself have teaching experience) frustrated the students with exercises that were too complicated.

Lesson 11: Understand your weaknesses and make contingency plans for dealing with them.

Unlike in the first project, we did not evolve the product with close interaction with a knowledgeable advocate for the product, or rather the evolutionary grain-size was far too coarse. Had we consistently trained someone from the evaluation team throughout the product development, they would have conveyed the appropriate level of exercise to us and helped with other misunderstandings.

But some of the problems perceived by the evaluators were simply errors of misunderstanding or overly-ambitious expectations. For example, they thought that it was necessary to develop the specifications in a particular order. In fact, this was not intrinsic to the approach, but merely an artifact of how the training session was presented.

They somewhat surprisingly thought the metadata databases should be built automatically, indicating a misunderstanding of their process on our part (violating the corollary to Lesson 10). We thought the redundancy of declaring what is desirable to collect vs. what will be collected by the survey was useful; we even designed an analyzer to check for this. Indeed, we could have done the declarations instead!

Another observation that indicates either a lack of understanding of the current system or an over-ambitious requirement on any system concerned automating updates:

> For example, if the user wished to make a change to the metadata, other parts of the system (e.g. form design and database) would occur automatically, or the user would be prompted to make relevant changes. The current system requires the user to remember to make changes in all places affected, thus making it prone to error.

The system actually provided analyzers to indicate where these changes needed to be made. The "view update problem" of multiple views of a system is not tractable. Their solution of interacting with the user is naive - they claim to want the roles of meta-data designer, questionnaire designer, and flow designer to be separate; yet here they would be asking one to do the others' work.

Lesson 12: Do not expect the domain experts to know what the computer can (should) do for them; do not expect them to understand what it cannot possibly do for them!

This is more a characterization of what type of person should be a DSL designer than anything else. One needs breadth and experience in other fields to do these designs. Harkening back to Lesson 11, if you do not have the breadth, enlist the help of someone who does, who may not have any expertise at language design.

> And, finally, one must not release software prematurely! Finally, technical difficulties with the software further heightened the frustration level of the users. ... Holding a test training session with a few users could have eliminated or reduced these problems.

Corollary to Lesson 12: Do not expect your users to overlook or forgive your design mistakes.

They will be very busy trying to understand how what works fits with their models. Asking them to overlook baroque interfaces and unnatural syntax that will be "cleaned up in the final product" may very well be too distracting.

To summarize, though, in the end the recommendations of the evaluators were really quite good. They proposed an orthogonal set of tools that could be adopted incrementally by the people with the appropriate expertise.

1. A questionnaire development tool using flowcharts. Develop a "front end" to CAI software to allow the use of flowcharts for developing survey instruments (an additional tool, not the only tool).
2. A utility to produce flowcharts from existing survey instrument code.
3. A survey development tool which would lead the user through the steps necessary to create a question, section, and/or entire instrument. This is envisioned as a "wizard" tool that would prompt the user for question text and conditions, fill text and conditions, universe for the question, etc., similar to the Prototyper described in Section B.1.
4. Integrate metadata into the survey development process. The data descriptions entered in the survey development tool described in #3 could be stored as metadata and used throughout the survey lifecycle (e.g. to generate paper questionnaires, in post processing/edits, to create file documentation, etc.)

Although further funding for a soundly engineered version of the system was denied, it is obvious that we had a very strong influence on the direction the SIC systems of the future will take.

6. Synthesis

It may help to organize the lessons above by considering what introducing a domain-specific language technology entails. Upon reflection, there are three types of issues that affect DSL technology design: technological, organizational, and social.

Consider first the technological issues. For researchers, these tend to be the most easily understood of the three. Generally, we are introducing new specification *artifacts*, usually in a newly designed *language* (taken loosely to mean concrete syntax, abstract syntax, database schema, or graphical style). Some set of *tools* is developed to produce and process the artifacts - variously parsers, GUIs, analyzers, generators, viewers, storage mechanisms - that assume the presence of a new *infrastructure* that is used to accomplish tasks that used to be done a different way. Generally, some of the tools will compile artifacts written in the new language into this infrastructure in order to effect the problem solving purpose. If there is any complexity at all to what is introduced, the *methodology* used to solve problems in the domain will need to be altered.

So in effect, just suggesting that someone use a DSL technology where none was used before is introducing as many as five intrusive innovations: artifacts, language, tools,

infrastructure and methodology. They have to be trained and feel comfortable with each of these innovations before they can again be effective in their problem solving domain; presumably much *more* effective.

Now most of this is not unique to DSLs: the introduction of any new technology may entail changes in these five technical areas. In fact, the hope for DSL designers is that, because the language they design will be close to one with which the domain experts are already familiar, the artifacts and languages will seem natural to them. But this is only half of the problem. How the new tools and infrastructure fit with their old ways of doing business is equally important. In fact, as proselytizers of specific tool and infrastructure technologies, we DSL designers are often somewhat careless in integrating our tools into their legacy methods and infrastructure, inventing baroque procedures they must follow to shoehorn the technology in.

I am now convinced that the key to overcoming the technical problems with the introduction of a DSL technology is to make sure that every step of the introduction process seems very simple and obvious to the affected community and occurs over a period of time, with considerable feedback along the way. Reject monolithic solutions and strive to produce functionality that can be incrementally adopted. Naturally, there are different situations into which DSL technologies are introduced. For example, the level of formal maturity of expression may be very low, as it was in all of the examples above; there, simply introducing a concise and precise way to formulate what they had been saying informally for years is itself an important, incremental first step. They should be (should have been!) trained and should become familiar with that before progressing with what might be done with artifacts in the new language.

Some clients will already be familiar with very well-defined languages, such as the NATO specification portion of the MTSL we used above, or people in problem domains with well-established mathematic models, such as in the control theory domain. There, the novelty will lie in the tools and infrastructure to be introduced by the DSL technology and the introduction of new tools may be more aggressive than initially desirable in a situation where no language existed in the first place, because they have less to learn. Again though, one must beware not to introduce artificial complexity into the problem solving process. For example, using a synthesizer and then implementing a manual process to integrate the synthesized output into a legacy framework - e.g. a "make" phase - may be very unfamiliar to an expert in the domain, but not in the implementation world. What we find to be an acceptable procedure as programmers can be very frustrating for non-experts in system matters.

A second source of major problems with technology introduction comes from organizational issues. Specific computing platforms may need to be supported, information ideally shared by two groups may need to be kept separate for security reasons, programming process models may be in place that need to be maintained for the contract to be filled, and who knows what legal hurdles may exist! Testing and injecting the technology may be made difficult by organizational standards as well. Often a parallel development effort (shadow project) is used to validate the approach; this may be simply too expensive or dangerous in some situations. Although these issues have the potential for impinging on the process greatly, I have little insight towards their solution!

Finally, social issues are often the most daunting of all problems when designing and introducing a DSL technology! There are a variety of personal relationship issues. You may be proposing to put someone out of a job - often many people. Without incredible incentives, perhaps for retraining into more advanced roles, you will be unable to get any help from those you displace regarding their expertise. You will do much better trying to leverage the expertise of someone bored with the amount of rote detail work required of the job. But beware, even an expert can be protective of such tedium - if using the DSL technology it only takes him or her one day to do what used to be done in ten, you are eliminating several copies of him or changing the nature of her job significantly. They may now become part-time resources! But most important of all, find a knowledgeable advocate within the organization that is both expert in the domain and the existing procedures and has some management credibility.

A second social phenomenon concerns the introduction of new technology into the methodology of the domain problem solvers: there may be several roles affected by the introduction. One should *avoid* this as much as possible, or more accurately, be sure to separate the concerns of each without impinging on the other. Our third experiment failed miserably in this regard, because we only designed the system for a single user, who apparently took on the roles of three different designers. A distributed systems approach would have been more amenable, with carefully separated functionality.

A third social phenomenon concerns the way the technology is initially sold to the client and later introduced into the working environment. *Demos are meretricious!* It is important to avoid creating overly-high expectations; naturally, this must be balanced with the need to generate high interest.

A final social phenomenon concerns yourself! Understand the roles you, the designer, will play: protocol observer (to see how things are done today), designer, specifier, engineer, training manual author, teacher, field expert in the domain itself! Recognize the areas you will be weak in and plan the technology development, introduction, and training, to minimize the impact of these weaknesses.

7. Previous Work and Final Remarks

Most of the work in DSLs that I am familiar with is concerned with the technological approaches and the unique situations in which they have been introduced [10, 16]. Little has been written on the problems with the introduction of technology into the working environment. Levy's attempt to motivate the use of domain-specific techniques [8] is a notable exception, and is the source of the 80% lesson above.

Fourth Generation Languages (4GLs) are another source of inspiration for DSLs. However, 4GLs try to be more "universal" than DSLs, or more accurately, tend to encapsulate a problem solving technique, rather than a problem domain. The leverage there comes about when problems are recast in terms of the problem solving technique, rather than adapting the technique to the problem domain. Nonetheless, Martin's treatment of 4GLs [9] is an excellent source of detailed advice on how to design for naturalness and "user-seductive" functionality.

I hope that the lessons learned can be of some value to others. Learning these lessons the hard way has cost us a lot of time and been the source of missed opportunity for further funding. Nonetheless, in fact, I believe these lessons tend to present an overly *pessimistic* outlook for the future of DSL technologies. For one thing, the user communities here were all non-programmers. Perhaps introducing DSL technologies to the programmers of tools to support the domain could be the first step in an adoption scenario, wherein communication between systems analysts and experts is facilitated by ever-increasing exposure of the domain experts to pieces of the DSL itself. DSLs embedded in the syntax of the programmers' language of choice might be more readily adopted; even a die-hard C programmer *might* find it difficult to ignore a code-expansion factor of 50! DSLs for computer-based technologies themselves are a likely candidate as well - robotics, syntax processing, device drivers, etc.; in fact, many conference papers have described successful DSLs in these areas [10], and the leveraged communities can be large. And remember also, many of the lessons are simply reflective of the difficulty of introducing any new technology. DSLs have the big advantage of naturalness to the domain experts.

In summary, I believe it is most important to maintain a conscious awareness of how technical, organizational and social issues are impinging on your DSL technology design and insertion process, but be confident that a DSL approach is often the best.

Acknowledgements

References

1. R. Balzer, M. Feather, N. Goldman, and D. Wile. Domain-specific Notations for Command and Control Message Passing. Internal report: USC/Information Sciences Institute, Marina del Rey, CA 1994. Note. contact authors at Teknowledge Corp., Marina del Rey, CA.

2. R. Balzer and N. Goldman. Mediating Connectors. In Proceedings of the 19th IEEE Conference on Distributed Computing Systems, Austin, TX, May 1999, 73-77.

3. C. Braun, W. Hatch, T. Ruegsegger, B. Balzer, M. Feather, N. Goldman, and D. Wile. Domain Specific Software Architectures - Command and Control. In Proceedings of the *IEEE Control Systems Society 1992 Symposium on Computer-Aided Control System Design (CACSD)*, Napa, California, March 1992.

4. D. Cohen. Automatic Compilation of Logical Specifications into Efficient Programs. In *Proceedings of the 5th National Conference on Artificial Intelligence -AAAI-86,* Philadelphia, Pennsylvania, April 1986, 20-25.

5. M. Feather. Language Support for the Specification and Development of Composite Systems. *ACM Transactions on Programming Languages and Systems* 9(2), April, 1987, 198-234.

6. N. Goldman and R. Balzer. The ISI Visual Design Editor Generator. *IEEE Symposium on Visual Languages*. Tokyo. Sep. 1999. 20-27.

7. R. Kieburtz, F. Bellegarde, J. Bell, J. Hook, J. Lewis, D. Oliva, T. Sheard, T. Walton, and T. Zhou. Calculating Software Generators from Solution Specifications. Technical Report # CS/E-94-032B of the Oregon Graduate Center. 1994.

8. L. S. Levy. Taming the Tiger: Software Engineering and Software Economics, Springer-Verlag: New York (1987).

9. J. Martin. *Fourth-Generation Languages: Principles.* Volume I. Prentice-Hall, Inc. 1985.

10. C. Ramming and D. Wile, eds. *IEEE TSE Special Issue on Domain-Specific Languages*, May/June 99.

11. A. Stratton. National Center for Health Statistics (NCHS) - Evaluation Report Chair. U.S. Census Bureau Evaluation Report: Survey Instrument Creator (SIC). Internal memo.

12. D. Wile. Local Formalisms: Widening the Spectrum of Wide-Spectrum Languages. Meertens, L.G.L.T. ed. *Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation*, Bad Toelz, North-Holland. 1986. 459-481.

13. D. Wile. Popart: Producers of Parsers and Related Tools, Reference Manual. USC/Information Sciences Institute, Marina del Rey, CA. 1993(a). (Contact author).

14. D. Wile. Integrating syntaxes and their associated semantics. Technical report. USC/Information Sciences Institute, Marina del Rey, CA. 1993(b). (Contact author).

15. D. Wile and R. Balzer. *Survey Instrument Creator (SIC) Training Manual.* Teknowledge Corp. 2000.

16. D. Wile, Supporting the DSL Spectrum. *Journal of Computing and Information Technology*. CIT 9, 2001 (4) 263-287.