Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 51 (2004) 265–290
Science of Computer Programming
www.elsevier.com/locate/scico

Lessons learned from real DSL experiments☆

David Wile*

Teknowledge Corporation, 4640 Admirality Way, Suite 1010, Marina del Rey, CA 90292, USA

Received 31 March 2003; received in revised form 1 October 2003; accepted 27 November 2003

Abstract

Over the years, our group, led by Bob Balzer, designed and implemented three domain-specific languages for use in real applications. Each was invented to "showcase" DSL language design and implementation technology that was the focus of our then-current research. 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. 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.

Keywords: Domain-specific language; Graphical language; Experience report; Program generation; Lessons learned

1. Introduction

We have undertaken three DSL experiments over the last 10 years. 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

☆ This is a revised and expanded version of [16] presented at HICCS, 2003.

* Tel.: +1-310-578-5350; fax: +1-310-578-5710.
E-mail address: dwile@teknowledge.com (D. Wile).

0167-6423/$ - see front matter
doi:10.1016/j.scico.2003.12.006

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. 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.

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" [15]. The lessons we learned, presented below, can usually be applied to these activities as well as to the purely syntactic or graphical DSLs.

In each case, *we were developing a technology for DSL design and implementation*, and, in essence, we were proselytizing for the approach. This differentiated us quite highly from someone who would have had the best interests of the organization at heart, i.e. someone tasked with finding the optimal solution to the problem, or even, someone tasked with the best DSL solution to the problem. Hence, the lessons might be most appropriately understood by DSL technology researchers and developers; some will already be well-understood by technology managers and decision makers. Nonetheless, even that group may benefit from seeing how DSL technology itself can impact their decisions.

Before beginning with the experiments, it might help to characterize the general problem of introducing technology into realistic applications, i.e. the technology transfer problem. Of course, one should expect problems with the technology/application match—and these are the main focus of the following discussions—but it can be especially surprising how organizational and sociological concerns impact the success of transfer. So, for example, independent of whether one is introducing a DSL-based technology or some other proposed solution, one might expect to find organizational reasons to resist the introduction—such as the technology's "fit" with a legacy development paradigm or system. Or even more likely, a sociological resistance to "doing things differently" will interfere. These three perspectives—technological, organizational, and sociological—will be used to organize the development of the lessons below.

First the experiments will be sketched in enough detail to establish context for the later presentation of the lessons learned. The lessons themselves will be introduced within a synthesis that should help to organize the approach to DSL technology design, development and adoption. Finally, some techniques for avoiding the various problems will be proposed.

2. The experiments

2.1. Military message experiment (MME)

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 in any way suitable for automatic validation 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—possibly even now—, 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 the "/" character, and was terminated with "//". Fig. 1 illustrates a simple snapshot from such a message.

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.

Part of the language was an adaptation of a formal document produced by NATO that describes the allowed fields and line sequence for each data-set type. Fig. 2 shows a snapshot of such a specification. We adopted it despite the fact that this notation was

[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].

```
GRAPH /PLAN/TRIANG/250833Z/AREA GREEN/15SUP398644/15SUP439640/15SUP403622//
```

Fig. 1. Message dataset example.

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

Fig. 2. MTSL type specification example.

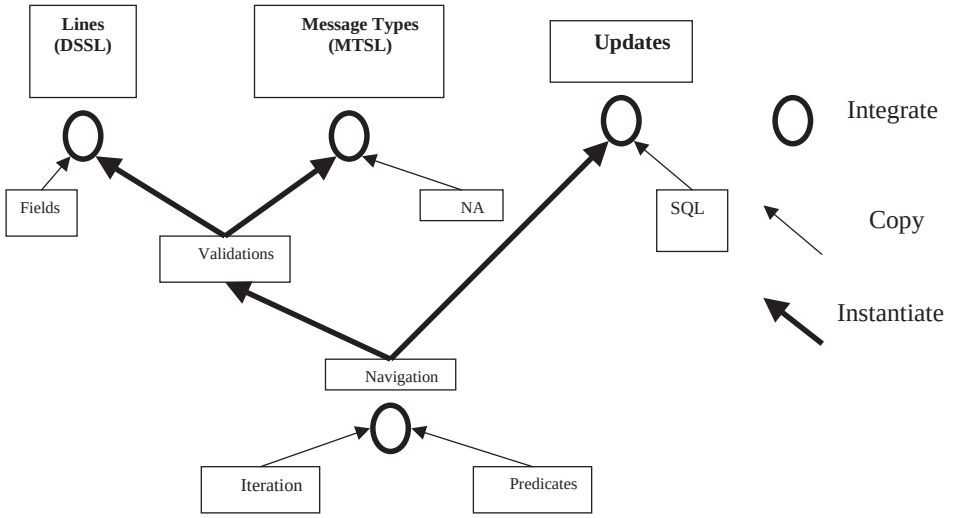


Fig. 3. C2 Language structure.

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 data sets—were never closed!

Fig. 3 represents the design of three separable languages, the DSSL, MTSL, and Updates languages. The languages were designed to maximize reuse of constructs in different contexts. Operators on grammars [14] were used to form the three languages. Here, "integration" of languages refers to their union. The "instantiate" arrows refer to parameterizations of the context free grammars. For example, although the Validations grammar is used in both the DSSL and the MTSL, their terminal symbols refer to fields or whole datasets, respectively. We were actually rather impressed with the apparent complexity of this set of languages, despite the simplicity of the component parts.

More will be said about language design for specific problem domains in the Lessons Learned section below.

It is important to mention that we did not cover the entire problem solution space with language constructs. 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.

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 schemas, thus guaranteeing a lack of run-time errors due to schema mismatch. We needed no separate schema declaration language within the DSL itself.

Moreover, our systematic approach allowed us to take advantage of run-time support that designers of the original code that was being replaced, failed to see. In particular, the languages were compiled into a data structure that was subsequently interpreted at run-time. Here, a message parsing mechanism was used to simplify processing that was normally included in each processing routine in an ad hoc fashion in the original implementations. Also, for each data set type one could precompile validation routines and invoke them for the specific message types and updates.

In the end, we were somewhat surprised to experience a 50 to 1 improvement in lines of generated code vs. lines of specification [1]! Admittedly, a more fair comparison would be to the source code we replaced, but we did not have that available for comparison. This issue also will be addressed more fully later.

Despite its success, the system never was critically reviewed for further development. Rather, a demonstration was given, the experiment was declared a success, and we participated in a brief follow-on project aimed at integrating the functionality with an existing message editing mechanism.

2.2. 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, dubbed "replenishment at sea" (RAS). 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 named for its role in guarding for man-overboard situations—was to move.

Both we and the experts in the domain sketched graphical charts to describe maneuvers by the ships. Hence, part of the language developed here was graphical.

(In fact, 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.)

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

Fig. 4 illustrates the initial arrangement of the ships, presented here for its "gestalt" nature rather than to be followed in detail. Similarly, Fig. 5 illustrates how the ships moved as part of the replenishment maneuver specification. Views such as these were augmented with declarative information about how maneuver steps were executed as expressed in Fig. 6.

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 replenishment at sea 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 that infrastructure that we assumed to be present.

Similarly to our experience with the Military Message System, the RAS system never was critically reviewed for further development. Rather, a demonstration was given, the experiment was declared a success, and we participated in a brief follow-on project aimed at applying the technology to tank formation specifications.

2.3. 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 [17]. 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. 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

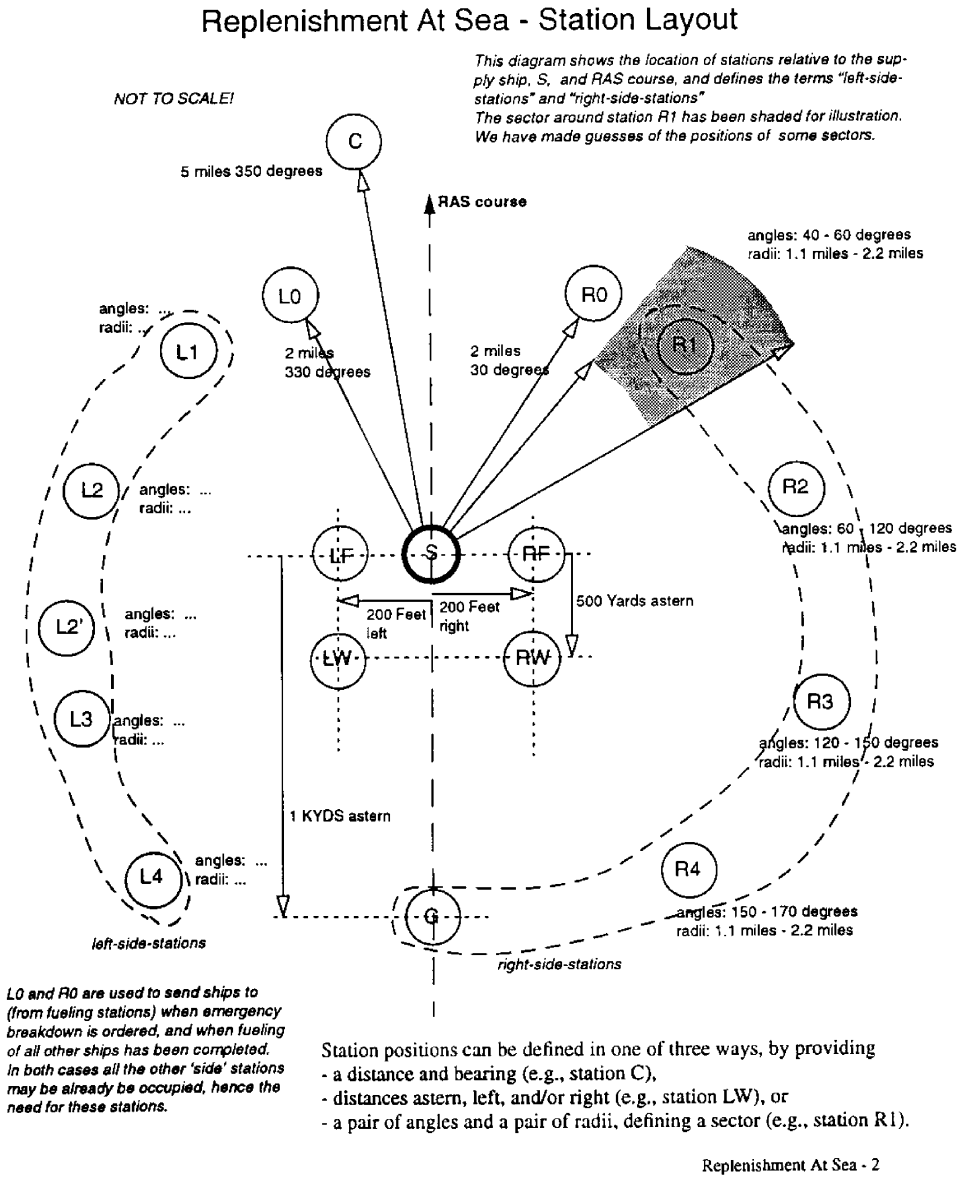


Fig. 4. The station layout view.

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.

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

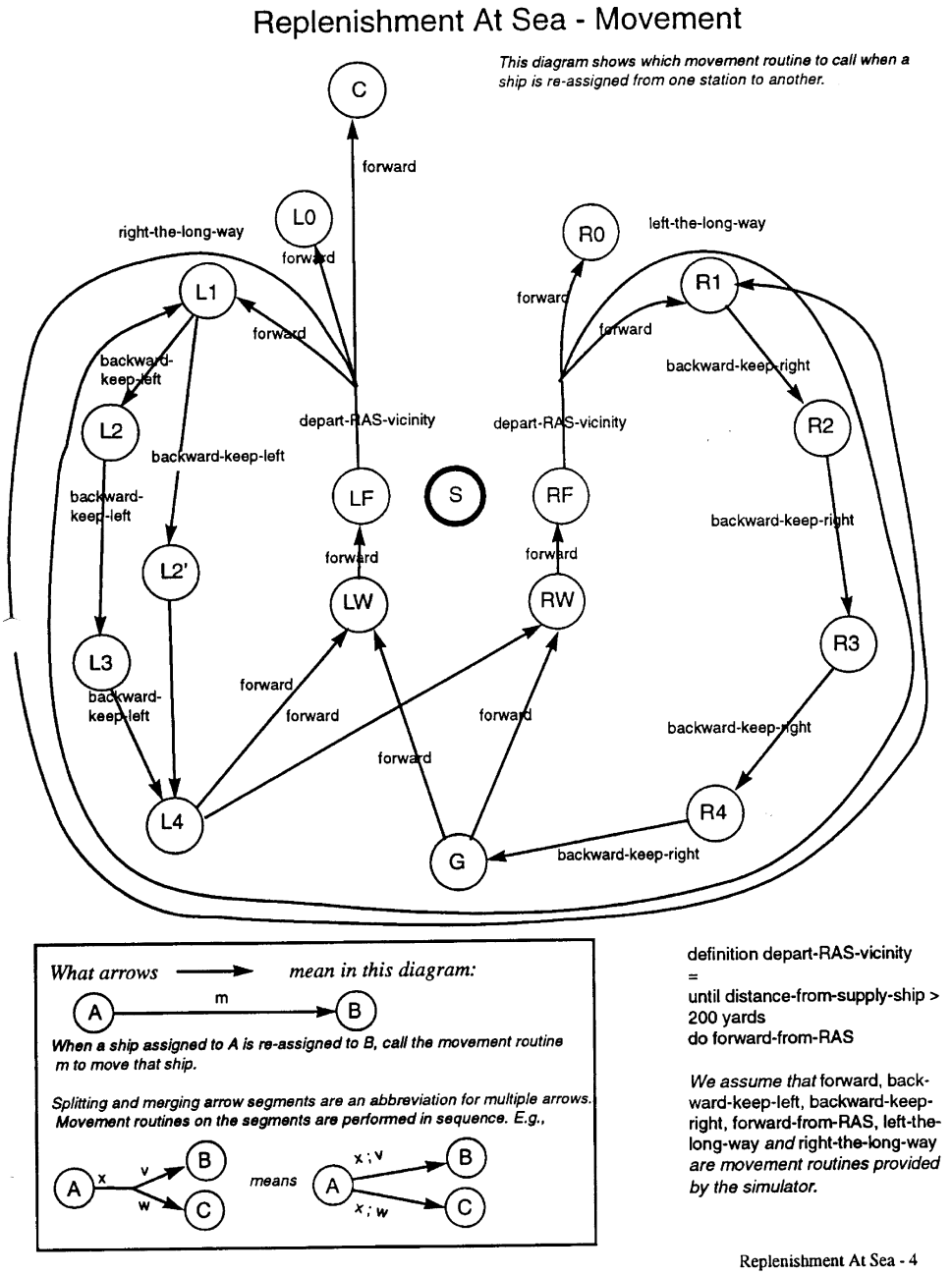


Fig. 5. The movement view.

```
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)
```

Fig. 6. Action and predicate definitions.

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 [6,15].

Within our extension to PowerPoint, called the Design Editor [6], the application designer designs a graphical interface, comprising icons and different kinds of arrows. The designer also declares what attributes may be used to decorate the various classes and superclasses of component and connector (icon and arrow) types. Domain types can also be enumerated. The resulting "domain design" takes the place of a syntax for the structural aspects of the domain. Specifications can then mix graphics with textual and graphical attributes in this hybrid DSL technology. Form-based attribute specifications allow one to express that an attribute can be optional, required and/or multiple valued. Its type can be prescribed and ranges given for numeric types.

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 Design Editor 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.

Fig. 7 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, Revisit-Survey 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

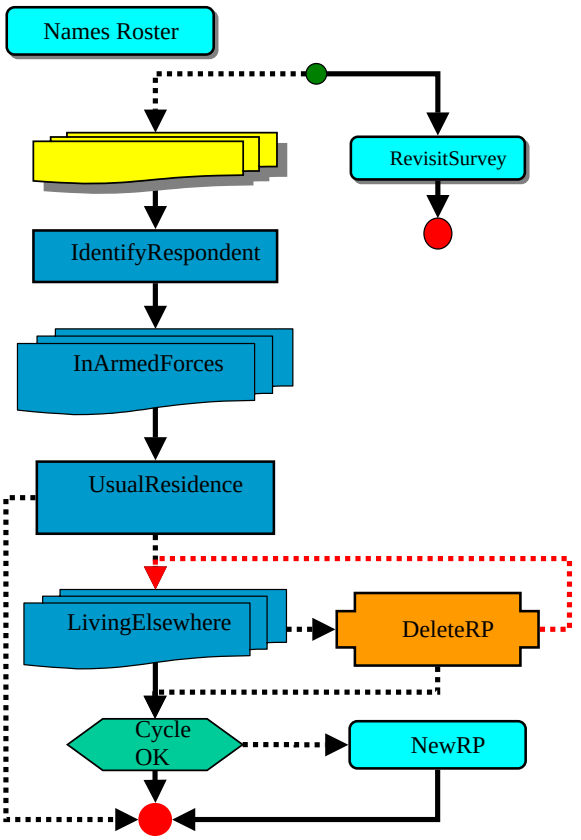


Fig. 7. A census survey flow-chart specification.

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.

The Design Editor allows the user to attach to the icons, 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 Fig. 8. Although all of the parts of the form are created using Tarantula, our tools imbued the different fields with extra semantics. The Instructions, Probe and Help frames are available for every form, allowing the designer

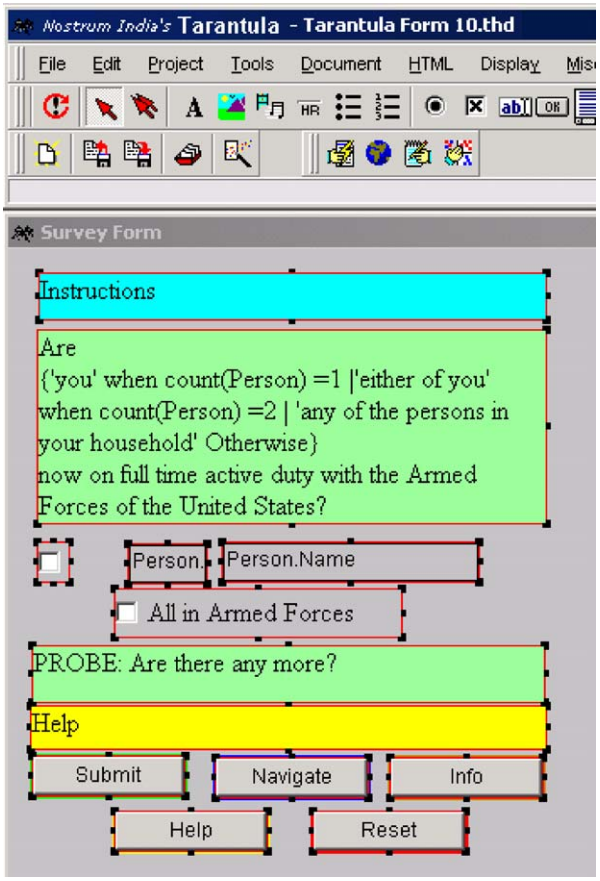
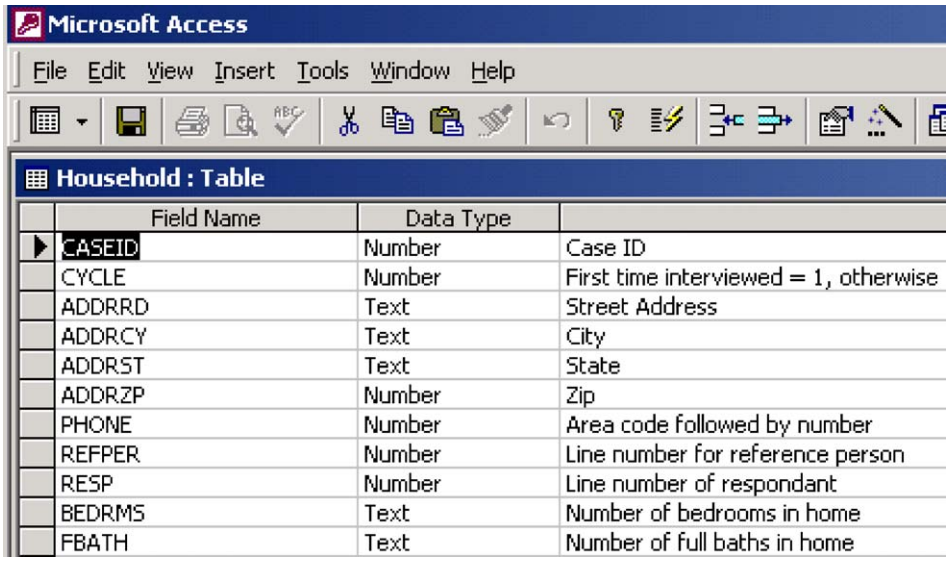


Fig. 8. A sample tarantula form.



Field Name	Data Type	
CASEID	Number	Case ID
CYCLE	Number	First time interviewed = 1, otherwise
ADDRRD	Text	Street Address
ADDRCY	Text	City
ADDRST	Text	State
ADDRZP	Number	Zip
PHONE	Number	Area code followed by number
REFPER	Number	Line number for reference person
RESP	Number	Line number of respondant
BEDRMS	Text	Number of bedrooms in home
FBATH	Text	Number of full baths in home

Fig. 9. Access definition of household rosters.

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. Some of the fields were implicitly repeated fields in the final forms that would be generated from this skeleton. In addition, our tools parsed the text in the normal text boxes to add variable interpretations. In this form, the prompt for the question 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 active duty with the Armed Forces of the United States" 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 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 Fig. 9.

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

As with the other experiments, the SIC was never developed further, although it underwent a serious evaluation process that will be described presently. It is safe to say that it had a greater impact on the funding organizations than the previous two had achieved.

3. 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 work-group 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 five 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. Below we consider the positive aspects. 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, and
- use of flowcharts to document the questionnaire.

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. The negative aspects will be taken up in the lessons section, following immediately.

To summarize, though, in the end the evaluators made recommendations for future development that 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...
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.

4. Lessons learned

It will help to organize the lessons we learned in the above experiments by considering how introducing a domain-specific language technology impacts technological, organizational, and social decisions.

4.1. Technological issues

Consider first the technological issues. For researchers, these tend to be the most easily understood of the three and they will almost certainly constitute the entrée into discussions with domain experts. The first of these lessons arose as a surprise reaction to our domain expert's behavior.

In the MME, we had an advocate for our (proposed) DSL technology in close contact with sympathetic experts in the domain. 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 (Fig. 2), rather than simply using the document as the specification!

Lesson T1: *Adopt whatever formal notations the domain experts already have*, *rather than invent new ones.*

The importance of this lesson was driven home very forcefully in the evaluation of the SIC, in the more corollary form:

Lesson T1 Corollary 1: *Use their jargon terms whenever possible.*

We insisted on using the term "metadata data item" for what they referred to as a "variable." (Their usage violated *our* jargon usage for variable.)

To quote the evaluation report again:

> 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.

Fortunately, we were able to adapt to the problem domain experts in the RAS experiment, where we introduced graphical notations from the very start. Their use of sketches was quite obviously going to be expressed extremely clumsily if done in a syntactic fashion. Hence,

Lesson T1 Corollary 2: *One should look to informal notations of the domain as the foundation for the DSL.*

And when no precedent notation or formalism can be discovered in the domain itself,

Lesson T1 Corollary 3: *Adopt conventional notations* (*lacking an in-place DS notation*), *rather than invent an idiosyncratic one.*

In the MME, for example, SQL was used to express the updates, rather than inventing a more streamlined language that was not already recognized by others in the database field.

A related lesson learned early on, was actually rather surprising:

Lesson T2: *You are almost never designing a programming language.* [2]

Most DSL designers come from language design backgrounds. There the admirable principles of orthogonality and economy of form are not necessarily well-applied

[2] Neil Goldman, a member of our group, was the first to enunciate this important precept.

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 MME's 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.

Lesson T2 Corollary: *Design only what is necessary. Learn to recognize your tendency to over-design.*

One must realize that spreadsheets are a success not because they allow programming, but rather in spite of the fact that they allow it!

Sometimes a DSL designer will feel exasperated and insist that programming language constructs be introduced. For example, in the MME MSTL 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. Instead, a facility was provided for the user to refer to externally defined predicates—to be provided by a programming expert, the operations engineer. Support for these predicates was simply beyond the expressiveness of the language.

The solution may lie in the following general guideline, presented here as a "lesson".

Lesson T3: *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]. 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. Again in the MME, for example, we were able to "type check" the SQL statements against separately specified, pre-declared database schemas, thus guaranteeing a lack of run-time errors due to schema mismatch. We needed no separate schema declaration language within the DSL itself.

The overview required of someone designing a DSL sometimes affords surprises in insights that simplify the overall problem solving tasks with the introduction of a little extra infrastructure.

Lesson T4: *Leverage the infrastructure you are providing to reduce the language complexity.*

Again in the MME, our systematic approach allowed us to take advantage of run-time support that designers of the original code that was being replaced, failed to see. As was mentioned before compilation of the languages into a data structure that was subsequently interpreted at run-time seems to be a very good way to leverage DSL technology. Additional structuring was provided in the message parsing mechanism, used to simplify processing normally included in an ad hoc fashion in the original implementations. The introduction of other specialized domain-specific components, precompiled for the specific message types and updates, provided leverage.

And, of course, in our later RAS and SIC experiments, we discovered the importance of not being bound into a one-size-fits-all approach to DSL design.

Lesson T5: *A mixture of specification techniques may be necessary to facilitate expression of the appropriate domain expertise.*

If we had insisted on a syntactic approach to these two domains we could not have begun to convince non-programmer domain experts to specify their solutions. The distance between things expressed well graphically and those best expressed syntactically is quite wide. In fact, the RAS made much more effective use of the graphical representations than the flow-chart representation of the SIC. Nonetheless, flow-chart, state-chart, dependency graph, etc., representations tend to attract advocates despite their near-equivalence to more compact syntactic representations.

4.2. Organizational issues

A second source of major problems with DSL technology introduction comes from organizational issues. A lesson learned early came again from the design of the MME language suite. We discovered that all three of the languages designed (Fig. 3) are used by different people in different organizations defining the specifications of messages, datasets, and DB updates. Hence, this normally good language design principle, of reusing like constructs for similar activities, was largely wasted in this situation!

Lesson O1: *Understand the organizational roles of the people who will be using your language.*

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 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.

Two related lessons were brought home in the SIC experiment. The failure to meet one of the objectives of the SIC is directly attributable to our misunderstanding of the census designers' expertise. 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. Hence, one must be sure to:

Lesson O1 Corollary 1: *Understand the background expertise of the people affected by the DSL technology introduction.*

Moreover, 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. Hence, it is important to:

Lesson O1 Corollary 2: *Understand the present solution design process thoroughly before undertaking to substitute a DSL approach.*

A second organizational lesson may have been more a misunderstanding on their part than ours, but it is clear that one must:

Lesson O2: *Be sure that the intended technology transfer process from your product into their organization's infrastructure is consistent with their business model.*

Again in the SIC experiment, another problem with COTS concerned the maintenance of tools based on them 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 the evaluation "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. This lesson covers a multitude of potential problems with inserting the technology into the organization; these solutions can be very difficult. This is an especially tricky problem for researchers who are essentially proselytizing a specific technology: it is really quite possible that their tools or methodologies cannot fit into the organization, and it is very important to discover this early on.

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.

4.3. Social issues

Finally, social issues are often the most daunting of all problems when designing and introducing a DSL technology! Almost certainly, the most important lesson here is to:

Lesson S1: *Find an advocate for your technology in their organization.*

In the MME we had found one but were not aware of how important an asset he was. Unfortunately, we did not have a strong domain-expert advocate for the RAS experiment and therefore we had to invent the algorithms for simulating ship movements, 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, but 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 f from one another. It turned out that the simulator took the object's longest dimension—the ship's length in this case—and rotated it about its center to determine the object's "boundary"!

Lesson S1 Corollary: *Establish close ties with a domain expert to produce the infrastructure that the system will be translated into.*

We did not really recognize how important having an expert/advocate for the new technology is until the third experiment, but it was clear after the RAS experiment how important the "expert" part is!

In fact, it is clear that the organizational misunderstandings that we had in the SIC experiment could easily have been remedied by closer ties with domain experts. Unlike in the first project, we did not evolve the product with close interaction with a knowledgeable advocate for our technology, or rather the evolutionary grain-size was far too coarse. Our interactions were more-or-less "demo driven" with fairly long periods between them during which little communication occurred.

Other lessons in this experiment 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. 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. Hence,

Lesson S2: *Understand your weaknesses and make contingency plans for dealing with them.*

But some of the problems perceived by the SIC 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 Corollary 2 to Lesson O1). 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! Hence, for some user communities:

Lesson S3: *Do not expect the domain experts to know what the computer can* (*should*) *do for them.*

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. [sic]

(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 S3 Corollary 1: *Do not expect the domain experts to understand what the computer 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 S2, if you do not have the breadth, enlist the help of someone who does, who may not have any expertise at language design.

And, as a final social lesson, 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.

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. That is,

Lesson S3 Corollary 2: *Do not expect your users to overlook or forgive your design mistakes.*

I believe that the most important lesson for DSL designers to take to heart is the guiding reason for DSL invention in the first place:

Strive for, *and expect*, *large improvements in expert productivity*!

Above, I mentioned that we were somewhat surprised to experience a 50 to 1 improvement in lines of generated code vs. lines of specification [1]! And again, a more fair comparison would be to the source code we replaced, but we did not have that available for comparison. But even if, by some strange quirk, we were generating 5 times too much code, the order-of-magnitude improvement cannot be dismissed easily. In addition, most arguments comparing hand-coding to compilation techniques are spurious, because one cannot normally *afford* to write that much optimized code by hand! To balance this, a variety of approaches can achieve some domain-specificity—such as the use of a set of carefully designed abstract data types [15], and comparisons to coding with such approaches will not likely yield such exuberant success.

At any rate, I believe that an order of magnitude improvement should be expected. But one must be aware of the risk. A variety of personal relationship issues may be problematical. 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. 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!

Another 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.

5. Suggestions

It is difficult to over-emphasize the importance of these simple observations. 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 languages they design will be close to those 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.

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.

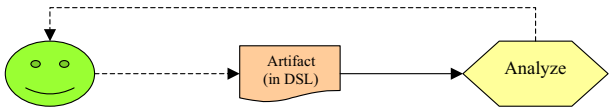


Fig. 10. Artifact, language, and tool.

It is unlikely that one can develop a universally useful checklist to ensure that the issues mentioned are avoided. It is especially unlikely that the order of issues considered can be prescribed. But I expect that it is important to cover issues broadly from each area before delving into too great a level of detail on any one.

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 sufficient time period to incorporate feedback along the way. Reject monolithic solutions and strive to produce functionality that can be incrementally adopted.

To see what might be possible, let us consider again the innovations the application engineers are being asked to learn and use: artifacts, language, tools, infrastructure and methodology. The simplest introduction of a DSL involves at least the first three of these as in Fig. 10. One should probably strive to introduce the application engineers to a single analysis tool as soon after the language is designed as possible. Comfort and acceptance of this will probably be the major hurdle to overcome, especially if there is no in-place application infrastructure that the DSL technology is replacing. It is also very likely to give you, in your role of technology designer, insight into the extent of familiarity with common interface idioms—e.g. the Microsoft tool look-and-feel—and into the ease with which the language and tool command elements can be assimilated by the engineer. In addition to becoming familiar with the language and the feedback from the tool, platform issues can be ironed out at this stage. It is often the case with DSLs that the tool support is provided on a different platform than the application infrastructure if one is already in place (since legacy systems tend to run on older mainframes and not personal computers), so progress here can be very important.

The difficulty of introducing even this much technology can vary widely. The two factors that will be most influential will be the extent to which formalisms are already part of the domain and the application engineer's familiarity with the tool platform. In all of the examples above the level of formalism was fairly low; there, simply introducing a concise and precise way to formulate what they had been saying informally for years was an important, incremental first step. They should have been familiarized and trained with that before progressing to what might be done with artifacts in the new language.

The early introduction of a tool, even a tool as simple as a parser with error message feedback, could have potential organizational and sociological advantages as well. It forces the issue of finding an advocate and, potentially, an expert domain engineer. (One can imagine that the advocate would be expert enough to act as application engineer, but not necessarily have the implementation savvy to fill the domain engineer's

role.) Moreover, this way organizational issues surrounding the choice of platform and interaction style must be faced early on.

One must balance the need to have an early introduction of the technology with the tendency to release unreliable code. (This exercise might also demonstrate just how fragile tool introduction can be, if done incorrectly!)

Fig. 11 illustrates the next logical step: adding a program generator to the tool mix. Not only will the additional functionality of the generator provide added complexity, but also the role it plays on a potentially different platform could be problematical. Again, if there is already an application in place that this is replacing, much of the complexity for the user will be reduced. The need to learn how to compile or assemble the application code will not arise. One should try to invoke as much of that functionality automatically. If possible, make invocation be an icon click in the initial tool stage development. What might seem to be a trivial step to you, the developer, can be just one more complication for an application expert.

Organizational issues likely to be encountered at this stage include misunderstandings about the application platform and misunderstandings about the process of inserting the application code into the application. For example, this could reveal that the intended frequency of regeneration was inconsistent with the allowable frequency. Perhaps there will be a design process requirement that the application be tested by another group in the corporation! Naturally, this kind of problem will have implications on the sociological relationships among the people in the different roles in the engineering process.

The final complications to be introduced are those that involve a second engineer, the operating engineer (e.g. the census survey taker in the third example) or perhaps a more complex infrastructure than the simple application infrastructure of Fig. 11. (If possible, one should try to hide such complexity from the application engineer, even if only a small part of the run-time system is generated.) Additional complexity might occur if there is yet another party whose responsibility it is, e.g., to install the software on the application platform. Once again, organizational issues might complicate introduction well beyond what appears to be simple.

Although this phase appears to be the final phase for technology introduction, it should probably merely be an intermediate phase in what now spirals back to the initial phase involving concerns with providing additional analyzers. One needs to establish a mechanism for choosing which analyzer to use and perhaps a process for using one or more analyzers before doing generation. In short, an entire methodology may need to be developed. In the limit, a process "wizard" might be called for, to keep the various players in line with the progress of design, analysis, compilation and installation of the new software.

Notice that although Fig. 12 separates out the roles of the application engineer (AE) and operating engineer (OE), it is still too simplified to reflect the complexity of our Survey Instrument Creator. Several analyzers were offered, some relying on the successful execution of others. An engineer would install the software on the survey takers' laptops. Moreover, several languages and processors were needed to construct the application. It is little wonder that our attempt failed: we presented the entire package and tried to educate people on how to use it in two days! And that is without even considering the considerable complexity of the domain itself!

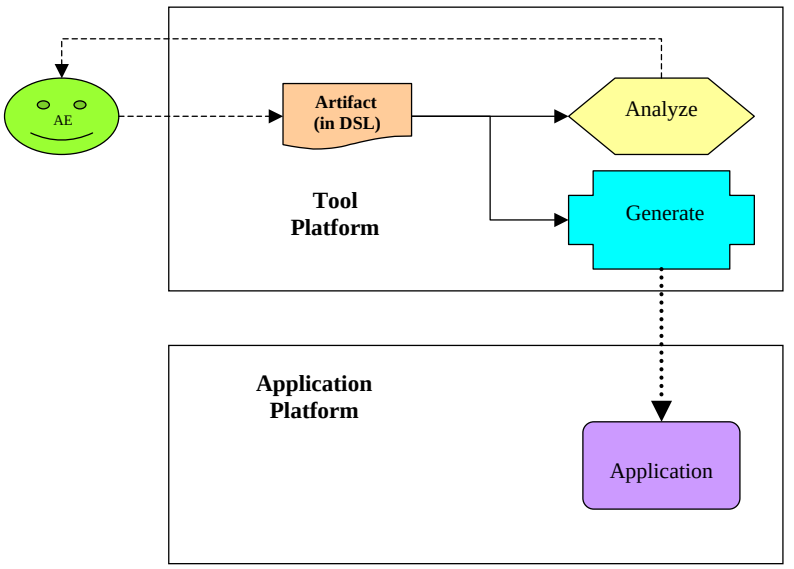


Fig. 11. Adding generator into application platform.

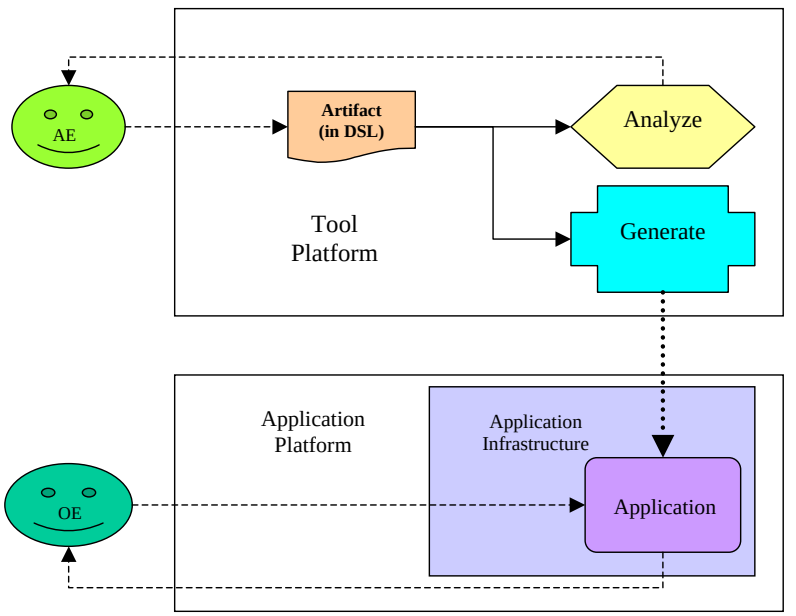


Fig. 12. Operating engineer and application infrastructure.

6. 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 [10,15]. 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. Introducing DSL technologies to the programmers of tools to support the domain can 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; a code-expansion factor of 50 is difficult to ignore! 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, introduce technology incrementally, and be confident that a DSL approach is often the best.

Acknowledgements

This work was sponsored by DARPA contract numbers F30602-93-C-0240, DABT63-95-C-0120, F30602-96-2-0224, F30602-00-C-0200, F30602-00-C-0218 and the NSF grant EIA/9876351. I would like to thank the many people who contributed to the designs and discussions of the DSLs represented, especially Bob Balzer, Neil Goldman and Martin Feather.

References

[1] R. Balzer, M. Feather, N. Goldman, 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, N. Goldman, Mediating connectors, in: Proc. 19th IEEE Conf. on Distributed Computing Systems, Austin, TX, May 1999, pp. 73–77.
[3] C. Braun, W. Hatch, T. Ruegsegger, B. Balzer, M. Feather, N. Goldman, D. Wile, Domain specific software architectures—command and control, in: Proc. IEEE Control Systems Society 1992 Symp. on Computer-Aided Control System Design (CACSD), Napa, California, March 1992.
[4] D. Cohen, Automatic compilation of logical specifications into efficient programs, in: Proc. 5th Nat. Conf. Artificial Intelligence—AAAI-86, Philadelphia, Pennsylvania, April 1986, pp. 20–25.
[5] M. Feather, Language support for the specification and development of composite systems, ACM Trans. Program. Languages Syst. 9 (2) (1987) 198–234.
[6] N. Goldman, R. Balzer, The ISI visual design editor generator, IEEE Symp. Visual Languages, Tokyo, September 1999, pp. 20–27.
[7] R. Kieburtz, F. Bellegarde, J. Bell, J. Hook, J. Lewis, D. Oliva, T. Sheard, T. Walton, T. Zhou, Calculating software generators from solution specifications, Technical Report # CS/E-94-032B, Oregon Graduate Center, 1994.
[8] L.S. Levy, Taming the Tiger: Software Engineering and Software Economics, Springer, New York, 1987.
[9] J. Martin, Fourth-Generation Languages: Principles, Vol. I, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.
[10] C. Ramming, D. Wile (Eds.), Special Issue on Domain-Specific Languages, IEEE Trans. Software Engineering 25 (3) (1999) 289–400.
[11] A. Stratton, National Center for Health Statistics (NCHS)—Evaluation Report Chair, US Census Bureau Evaluation Report: Survey Instrument Creator (SIC), Internal memo.
[12] D. Wile, Local Formalisms: Widening the Spectrum of Wide-Spectrum Languages, in: L.G.L.T. Meertens (Ed.), Proc. IFIP TC2/WG 2.1 working conf. on program specification and transformation, Bad Toelz, North-Holland, 1986, pp. 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, Supporting the DSL Spectrum, J. Comput. Inform. Technol. CIT 9 (4) (2001) 263–287.
[16] D. Wile, Lessons Learned from Real DSL Experiments, in: Proc. Hawaii Internat. Conf. on Computer Sciences (HICCS), DSL track, Kona, 2003.
[17] D. Wile, R. Balzer, Survey Instrument Creator (SIC) Training Manual, Teknowledge Corp., 2000.