JARGONS FOR DOMAIN ENGINEERING

Lloyd H. Nakatani, Mark A. Ardis, Robert G. Olsen, and Paul M. Pontrelli

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Jargons for Domain Engineering

Lloyd H. Nakatani, Mark A. Ardis,
Robert G. Olsen, Paul M. Pontrelli
Lucent Technologies USA

Abstract

In the Family-oriented Abstraction, Specification and Translation (FAST) domain engineering process for software production, a member of a software product family is automatically generated from a model expressed in a DSL. In practice, the time and skill needed to make the DSLs proved to be bottlenecks. FAST now relies on jargons, a kind of easy-to-make DSL that domain engineers who are not language experts can quickly make themselves. We report our experiences with jargons in the FAST process, and describe the benefits they provide above and beyond conventional DSLs for software production and other purposes.

1. Introduction

We report here our experience with jargons [Nakatani97] for software engineering. Jargons are DSLs that are unusually easy to make. We use jargons within the framework of the Family-oriented Abstraction, Specification and Translation (FAST) domain engineering process [Parnas76] [Cuka98] to automate software production. Previous attempts with FAST had foundered when the DSLs needed for domain modeling took too long to make. Because jargons can be made quickly and easily, they seemed a good alternative to conventional DSLs.

Preliminary experiments made us optimistic that jargons would work for FAST. We made jargon equivalents of two existing DSLs that had each taken over a year to make, even using language implementation tools such as `yacc` [Johnson75]. The results were dramatic. Each jargon took less than a week to make. The catch was that the jargon maker in the experiments was the inventor of jargons. Further work was needed to see if jargons could be made as quickly by software developers who were domain experts but were not experts in language design and implementation, and had no prior experience with jargons. We believe that jargons are most likely to succeed when developers can make their own.

Our effort was focused on getting answers to the following questions:
- Can jargons handle the complexity of real world domains?
- Can domain experts make their own jargons?
- Do jargons avoid the major pitfalls of DSLs?

Of course, we were open to unexpected discoveries, good and bad.

The FAST process is described in section 2, jargons in section 3, the software domain in section 4, benefits of jargons in section 5, potential pitfalls of DSLs in section 6, related work in section 7, and conclude with lessons learned in section 8.

2. FAST Process for Domain Engineering

Many software products constitute a family consisting of many variations of essentially the same thing. The variations may be successive generations of a product, or different versions of a product for different platforms, customers, or market segments. A familiar example of a hardware family is a car that comes in stripped-down or luxury versions and in a choice of two- or four-door models. We believe that software families are ubiquitous. However, good examples are hard to come by, because software is usually not described in those terms. An example from telecommunications is the 5ESS(RM) electronic switch software [Martersteck85] that comes in different versions customized for different customers and hardware configurations.

FAST exploits the properties of software families to make the production of family members more efficient. The FAST process is split into two phases: domain engineering, and application engineering. (See Figure 1.)
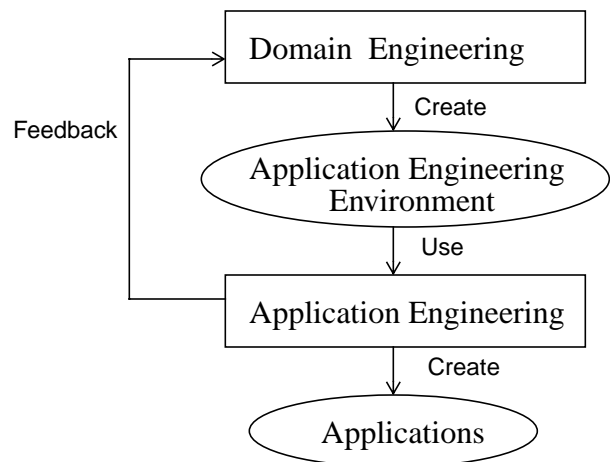

Figure 1. FAST process of domain engineering

From the description below, one might get the impression that FAST is a serial process, but in fact it is an iterative process in practice with much feedback between the phases.

In the *domain engineering* phase, family members are analyzed to discover their commonalities—what is the same about every member—and their variabilities—what differentiates one member from another. A software architecture reflecting the commonalities and variabilities is then designed.

The parts constituting the commonalities are expected to remain constant over family members. There is little need to optimize their production, since they will only be created once. On the other hand, integration of these parts with other software is a concern. Some FAST projects elect to automate the generation of commonalities code in order to simplify later integration.

The variabilities are addressed in the domain engineering phase by making one or more jargons to model family members solely in terms of their variabilities. Translators are written to translate models into variabilities code. The use of high-level modeling languages, of which jargons are instances, distinguishes FAST from other domain engineering processes.

In the *application engineering* phase, a model of a particular family member is written in terms of one or more jargons, and then translated automatically into variabilities code. Finally, the commonalities code and variabilities code are integrated to produce the product. The integration process, for instance, may be building a load module incorporating the compiled commonalities and variabilities codes.

Two aspects of FAST have important ramifications for jargons. First, partitioning a product into its commonalities and variabilities is key to making automation feasible. The commonalities usually represent the core of the product and encapsulate most of its complexities. The commonalities code and variabilities code are separated, allowing for independent development. The variabilities part is usually much smaller and simpler than the commonalities, so automating its production is easier than automating the production of the entire product. Second, the variabilities often represent distinctly different features of the product, and may not lend themselves to modeling with a single jargon. When this is the case, several different jargons may be necessary to model the distinct variabilities.

3. Jargons: Domain Engineered DSLs

In essence, jargons are domain engineered DSLs. The expected benefit is a production environment called InfoWiz for making jargons efficiently. With InfoWiz, we can make a jargon in a matter of days or weeks instead of months or years for conventional DSLs. Some unexpected benefits are: 1) jargon making simplified to a Do-It-Yourself (DIY) activity; 2) composable jargons (i.e., different jargons are compatible and can be used in combination); and 3) multipurpose models (i.e., multiple products can be produced from a single model written in a jargon). So compared to conventional DSLs, jargons provide more benefits at less cost.

Jargons comprise a family of custom-made DSLs with the following commonalities:
- Abstract syntax: Every expression of every jargon has the same abstract syntax.
- Generic interpreter: All jargons are processable with the same generic interpreter specialized at runtime with the semantics of the pertinent jargons.

A jargon is distinguished from another by the following variabilities:
- Concrete syntax: A set of concrete expressions
- Semantics: A set of actions corresponding to the expressions that define a semantics of the jargon

InfoWiz, the application engineering environment for making jargons, consists of the following components:
- WizTalk abstract syntax
- InfoWiz generic interpreter
- Fit programming language for defining actions
- API functions for interfacing actions to the InfoWiz interpreter

This section describes these components to show what jargons are like, and how they are designed, implemented and used.

3.1. WizTalk Abstract Syntax

The WizTalk abstract syntax prescribed for every expression of every jargon is

```
;term(note-1| ...|note-N)[memo]
```

The `;` metacharacter is a *marker* that distinguishes jargon expressions from *plaintext*, which is any text that is not a jargon expression. The marker makes possible *markup jargons* with expressions embeddable in plaintext; for example

```
The ;cb[;] metacharacter is a
;i[marker]
```

This is the beginning of this paragraph in the markup jargon used to format a draft of this paper.

The *term* is the name of the expression. The jargon

designer is free to choose concrete terms that best reflect the natural terminology of the domain. If care is taken to make the concrete terms unique across all jargons that might collaborate, the jargons are composable without conflict. The term is case-sensitive, and `.` and `_` can be used interchangeably as separators between words of a multi-word term.

The *memo* is the information that is the focus of the expression. WizTalk allows three syntactic variants of a memo. When the memo fits on a single line, the preferred variant is an *inline memo*:

```
;author[Mark Twain]
```

The `[` and `]` metacharacters are *memo delimiters*. When the memo consists of multiple lines, one variant is a *block memo*:

```
;address[
123 Main Street
Anyville, NJ  01234
]
```

where the memo and its closing `]` delimiter are aligned with the margin established by the indentation of the line containing the expression. An alternative variant for a multi-line memo is an *inset memo*:

```
;address
    123 Main Street
    Anyville, NJ  01234
```

where the memo is tab-indented one level deeper than the indentation of the line beginning the expression. Indentation is thus syntactically significant. Inset memos make the hierarchical structure of complex models easy to see.

The *notes* are either attributes of the memo, or parameters to control the processing of the expression. For example, in

```
;state(END)
    ;exit
```

the note is the name of the state of a finite state machine. The `(` and `)` metacharacters are *note delimiters*, An expression may have neither note nor memo; the `;exit` expression above is an example.

The WizTalk abstract syntax has proven versatile enough for markup jargons to model the format of documents, data jargons to model hierarchically structured information, and programming jargons to model algorithms.

3.2. InfoWiz Generic Interpreter

The InfoWiz generic interpreter can process any jargon when customized with a semantics of the jargon. A semantics of a jargon is defined by a set of *actions*, one for each expression of the jargon. An action, which is a function written in the Fit programming language, specifies how the information associated with an expression should be processed. A file containing a set of action definitions is called a *wizer*. The InfoWiz interpreter, which is also written in Fit, customizes itself by incrementally loading one or more wizers, and automatically integrating the actions they contain.

Once customized, the InfoWiz interpreter processes a model written in the jargon by parsing the model; traversing the parse tree in top-down, left-right, depth-first order; executing the action corresponding to the expression at each node of the parse tree in traversal order, and appending the expression's product to an output buffer. Plaintext is a degenerate expression whose product is the verbatim text. A scalar product for a parent expression is produced by concatenating the products accumulated in the output buffer for the child expressions in the memo of the parent expression.

Actions are described in more detail later, but a tiny example is presented here to illustrate how the InfoWiz interpreter is used. This jargon document in file `greet.doc`

```
;greet[InfoWiz]
```

consists entirely of the `;greet` expression. This action in the wizer file `hello.w`

```
A_greet
    WizOut "Hello, " GetWizMemo
```

is a Fit function that defines a semantics for the `;greet` expression. The action name consists of the `A_` prefix followed by the expression's term. The `GetWizMemo` API function processes the memo of the expression, and returns the product, which for this example is the plaintext `InfoWiz`. The `WizOut` API function splices together its arguments and appends the result to the output buffer. The `wiz` command

```
$ wiz -w hello.w greet.doc
    Hello, InfoWiz
```

runs the InfoWiz interpreter that integrates the actions in the `hello.w` wizer, processes the document in `greet.doc`, and writes the product to the standard output. The `-w` command option identifies `hello.w` as the wizer.

3.3. Fit Programming Language and API Function Library

The Fit programming language makes it possible to write actions quickly with the support of the API function library. Fit is a high-level, general-purpose, interpreted programming language developed at Lucent and AT&T. It is beyond the scope of this paper to explain Fit. Suffice it to say that Fit has excellent facilities for text processing, supports multiple programming paradigms including object-oriented, and has a source code debugger integrated into its interpreter.

The API functions, which are written in Fit, enable actions to interface with the InfoWiz interpreter. The API library consists of less than 50 functions. Of these, about a dozen are frequently used.

3.4. Multipurpose Models

Because actions are easy to write and their integration into the InfoWiz interpreter is automated, it has proved practical for a jargon to have multiple semantics. The consequence has been multipurpose models capable of producing many different products. If yet another product is needed from a model, it can be produced simply by writing a new wizer and processing the model with the wizer.

We use this greeting model from before

```
;greet[InfoWiz]
```

to illustrate how multipurpose models work in practice, We define another semantics for the ;greet expression with this action:

```
Macro HELLO ~[
main()
{
printf("%s", \
    "Hello, <greetee>\n" );
}
]~

A_greet
    WizOut Change "<greetee>"
(GetWizMemo) HELLO
```

When the greeting model is processed with this new semantics, the result is a C program that prints Hello, InfoWiz:

```
$wiz -w Chello.w greet.doc
main()
{
    printf( "%s", "Hello,
InfoWiz\n" );
}
```

Multipurpose models ensure that all the products generated from a model are consistent with each other. For example, if the greeting model is changed to

```
;greet[World]
```

this change is reflected in all the products obtained from the model.

3.5. Composability of Jargons

Because all jargons have a common abstract syntax, multiple jargons can be processed by the InfoWiz interpreter simultaneously customized with the semantics of the jargons. In effect, this makes all jargons composable and able to collaborate with each other to solve problems beyond their individual reach.

To illustrate jargon composition in practice, we use this finite state machine model of an interactive dialogue:

```
;state(START)
    ;next.state[INPUT]
;state(INPUT)
    ;=x[;input[Type name: ]]
    ;if( ;.x == q )
        ;next.state[GOOD BYE]
    ;else
        ;next.state[HELLO]
;state(HELLO)
    ;output
        ;frame
            ;star[Hello, ;.x]
    ;next.state[INPUT]
;state(GOOD BYE)
    ;output[Good bye]
    ;next.state[END]
;state(END)
    ;exit
```

This model is composed of four jargons: the *FSM*, *Base*, *Flow*, and *Banner* jargons: The *FSM* jargon for modeling finite state machines consists of the ;state, ;next.state, and ;exit expressions. The *Base* jargon, which comes standard with the InfoWiz interpreter, includes the ;=x[InfoWiz] expression to set variable x to InfoWiz, and the ;.x expression to get its value, among others of similar generic nature that are likely to be useful for many domains. The *Flow* jargon for modeling algorithms includes flow control expressions such as ;if and ;else, the ;input and ;output expressions, and relational predicates using infix notation such as ;.x == q to test whether variable x has value q. The *Banner* jargon for modeling "banners" such as

```
=============================
|  ***  Hello, InfoWiz  ***  |
=============================
```

includes the `;frame` expression to put a frame around a message, and the `;star` expression to bracket a message with `***`.

Because the model is composed of multiple jargons, multiple wizers are needed for its execution:

```
$ wiz -w flow -w fsm.w \
      -w banner.w hello.doc
Type name: InfoWiz
============================
| ***  Hello, InfoWiz  *** |
============================
Type name: q
Good bye
$
```

The results of the execution is an interactive dialogue shown above.

To illustrate how easy wizers are to write, shown below are the wizers for two of the jargons. This is the wizer for the *FSM* jargon

```
A_state label
   Local fsm
   Set fsm[label] GetWizTree

A_next_state
   Local fsm state
   Set state fsm[GetWizMemo]

A_exit
   Local state
   Set state nil

WizAfter out
   Local fsm state
   Set state fsm["START"]
   While state
      WizMemo state
```

This is the wizer for the *Banner* jargon

```
A_star
   WizOut "***  " \
      (GetWizMemo) "  ***"

A_frame
   Set message Splice "| " \
      (GetWizMemo) " |"
   Set edge<Thru 1 $message> \
      "="
   WizOut |"\n"| edge message \
      edge
```

For the lack of space, we provide no further explanation of the wizers.

4. Configuration Control in the 5ESS

The 5ESS is a highly reliable telecommunications switch. The reliability stems in part from redundant hardware and automatic reconfiguration software that removes faulty hardware units from service and replaces them with their spares. The Configuration Control domain, which includes this reconfiguration software, was re-engineered with the FAST process to speed up software production. Our report is based mostly on our experience with jargons in the Configuration Control domain.

An analysis of the Configuration Control domain resulted in a clean separation between the configuration and the algorithms that change the configuration. The configuration is a variability of the domain since different switches are composed of different hardware units, and the units can be in different conditions (e.g., *ready* or *working*). The reconfiguration algorithms are operations that change the configuration. They are the commonalities of the domain, since the algorithms are the same for all units.

Per the FAST process, a Configuration jargon was made to model a configuration. A configuration was representable as a graph with nodes corresponding to the hardware units, and edges representing the relationship between the units. A wizer was written to translate a configuration model into declarations of populated C data structures that represented the hardware configuration of a switch in a form suitable for the reconfiguration algorithms. To facilitate the translation, a Cstruct jargon was made to model the C data structures. The translator was modeled by the composition of both models: the configuration model for the source, and the C data structure model for the target.

In order to separate the common parts from the variable parts of the reconfiguration algorithms, a Controller jargon was made. In addition to separating commonalities from variabilities, this jargon made it possible for other domain experts to check the accuracy of the algorithms. Each algorithm was modeled in the jargon, translated into a finite state machine, and from there to C code. When a bug was found in the algorithm code, its model was fixed, and new code regenerated. From jargon models totalling about 500 lines, over 50,000 lines of C code were generated.

In the final integration process, the codes for the configuration data structures and the reconfiguration algorithms were compiled together along with handwritten code that handled the interface between the two, and also between the Configuration Control domain and other domains.

5. Benefits of Jargons in Domain Engineering

Jargons provide many benefits to the FAST domain engineering process. In this section we describe attributes of jargons that we believe were key to their success in our environment.

5.1. Domain Decomposition and Modeling

Jargons fostered a decomposition (divide-and-conquer) strategy for coping with the complexity of a domain. For the Configuration Control domain, the first decomposition divided the domain into its commonalities (reconfiguration algorithms) and its variabilities (configuration) per the FAST process. The configuration subdomain further divided into the configuration itself and the C data structures into which the configuration had to be translated. Three jargons were made in accordance with this decomposition: the Controller jargon for modeling the reconfiguration algorithms; the Configuration jargon for modeling the configuration; and the Cstruct jargon for modeling C data structures. The Configuration jargon was a natural by-product of the domain analysis, and bears a strong resemblance to a section of the analysis document. Very little effort was required to design this language. Similarly, the Cstruct language was modeled on existing data structure facilities of the C language, so it was easy to design.

Two other domains—the Call Billing and Measurements domains—re-engineered with the FAST process also decomposed naturally into multiple subdomains. The Call Billing domain produces software that generates accounting records for billing calls. This domain decomposed into four subdomains: 1) algorithms that determine the kind of record to be generated for a call, 2) the format and contents of the fields comprising a record, 3) the concrete values of abstract variables that could populate a field, and 4) the attributes of the abstract variables. A separate jargon was made for modeling the artifacts of each subdomain. The Measurements domain produces the software that reports on the performance of equipment in a telephone office. The domain has already decomposed into four subdomains, with more expected. Three of the subdomains have their own jargons, and the fourth was further decomposed into three subdomains with their own jargons. So altogether, there are six tiny jargons for modeling the subdomains of the Measurements domain.

Decomposition works only if the solutions for the parts can be composed into a solution for the whole. For our approach, this means that the subdomain models expressed in different jargons must be composable into larger models for the entire domain. Model composition was the rule in every domain. In the Configuration Control domain, the configuration and reconfiguration algorithm models were composed to build a simulator to test the algorithms on real data. In the Call Billing domain, all the models had to be composed in order to generate the records. And in the Measurements domain, models of the measurement, report content, and report structure had to be composed to generate the reports.

5.2. Do-It-Yourself Jargons

With InfoWiz, a jargon is so easy to make that domain experts can make their own. Such Do-It-Yourself (DIY) jargons can be expected to provide the following benefits:

- Better jargons
- Lower risk
- Easier maintenance

A DIY jargon made by a domain expert is likely to turn out better than one made by a language expert who is not a domain expert for several reasons. First and foremost, the domain expert knows the domain: the "natural jargon" spoken among experts, the subdomains of the domain, what has to be modeled in each subdomain, how the models should be composed, what products have to be generated, the assumptions needed to translate models into products, the legacy languages of the products, and how the generated products integrate with other code, such as the commonalities code. Second, the domain expert and the end-users of the jargon usually belong to the same organization. The easy communication between the jargon maker and jargon users makes for an ideal rapid prototyping environment. Jargons lend themselves to rapid prototyping because of the ease with which they can be made and modified. The domain expert can take good advantage of the environment and rapid prototyping to make a "user friendly" jargon.

A DIY jargon is less risky. When the organization makes and supports its own jargons, the risk that comes from depending on outside experts is eliminated. Bugs can be fixed immediately, and new features added as needed, no matter how minor. In event of a crisis years down the road when the outside expert is no longer available, the organization will have the wherewithal to cope.

Compared to general-purpose programming languages, jargons can be expected to evolve rapidly, which puts a premium on easy evolution. A jargon is custom-made for modeling a particular domain, so as the domain evolves to meet changing needs, its jargon must evolve to keep up. It follows that jargons must be easy to maintain if they are not to become quickly obsolete [VanDuersen97] [Spinellis97].

Jargon making is indeed simple enough to be a DIY activity. In the Configuration Control domain, the jargons were created by a team of three domain experts and one InfoWiz expert. The domain experts produced and maintain all of the jargons. Three completely different versions of one jargon were prototyped in a month. In the Call Billing domain, the jargons and translators were the result of collaboration between an InfoWiz expert and a domain expert. However, another domain expert added major new capabilities to one of the jargons. In the Measurements domain, all of the jargons were made by domain experts with an InfoWiz expert serving as a reviewer only. None of the domain experts had any previous experience in making languages.

DIY jargons are possible because most of the hard work is already done. Skill in the art and science of syntax design and grammar specification is unnecessary because WizTalk prescribes a ready-made abstract syntax for all jargons. Skill in the art and science of building a parser and interpreter is unnecessary because the ready-made, generic InfoWiz interpreter works for all jargons. Defining the semantics of a jargon is reduced to writing a set of actions in a high-level, interpreted programming language. No work is necessary to integrate the actions into the interpreter because the integration is automatic. In the Configuration Control domain, the domain experts mastered jargon technology within a few days.

5.3. Raising Software Quality

Jargons go beyond conventional DSLs to improve the quality of software products. Jargons share with conventional DSLs the benefits of specialization, high-level abstractions, and automatic generation to eliminate accidental errors. Jargons go beyond these conventional techniques to improve software quality by the following means:

- Simplification through decomposition of a complex domain into subdomains
- Generation of related subproducts from a single source to ensure their consistency
- Transformation of models into more readable forms for review
- Automatic model checking with user-defined types

The decomposition of a complex domain into simpler subdomains, and modeling each in its own jargon, improves quality indirectly. The simpler the domain, the simpler their jargons, and the simpler the jargons, the more likely are their designs and translators to be correct. Moreover, simple models expressed in simple jargons are shorter and easier to write, review, and debug. In all of the domains we've worked on, decomposition

has resulted in simplifications that most likely improved the quality of the final product. However, without metrics of complexity or simplicity, it's hard to attribute any improvement in quality to the simplification that comes from decomposition.

One measure of the quality of a product is the consistency between its subproducts. Consistency is ensured if all the subproducts are produced from a single multipurpose model. Multipurpose models were used to good effect in the reconfiguration subdomain of the Configuration Control domain. Two human-friendly representations of each reconfiguration algorithm were generated from its model: a pictorial representation as a flow chart, and a stylized English representation. Reviewers preferred the pictorial and natural language representations because they were easier to comprehend. The fact that the review representations and the software were generated from the same model ensured that the product approved by the reviewers was what was actually produced. In addition, the graphical tool used by application engineers was enhanced with a simulator that showed the behavior of the reconfiguration algorithms. The original models of the algorithms were translated by a wizer into an internal finite state machine model in the language of the graphical tool. This guaranteed that the behavior of the simulator would remain consistent with the generated code.

A specification of a jargon can be written in a ready-made Checker jargon that comes standard with InfoWiz. Such a specification expressed is the analog of a Document Type Description in SGML [Goldfarb90]. A specification is translated into a wizer that defines a self-checking semantics for the jargon. When a model is processed with this semantics, each expression checks itself against its specification. No specification was written for the jargons of the Configuration Control domain because the Checker jargon was not available at the time. In retrospect, specifications of the Configuration Control jargons would have been of limited use. Configuration models are now being generated by an interactive tool with a graphical user interface (GUI) that prevents the kinds of mistakes that would be caught by a model checker. The tool makes a checker of little value. Algorithm models require analysis and checks of its dynamic behavior. A specification would only do a static analysis and checks that would be of limited value in detecting bugs in the algorithms. The value of a specification goes beyond checking because it also serves as documentation for users of a jargon.

6. Avoiding DSL Pitfalls with Jargons

Some pitfalls lie in the path leading to the widespread use of DSLs. Unless these pitfalls are successfully avoided, we feel that the use of DSLs will prove counter-productive in the long run. Jargons avoid these pitfalls.

6.1. Pitfall: Balkanized Domains

Conventional DSLs are not composable. As a consequence, different DSLs cannot work together just as Lisp, C, and Java cannot. DSLs make problems in their respective domains easy to solve, but their incomposability will make problems involving multiple domains even harder to solve by preventing collaboration between domains that are each part of the solution. It is critical to avoid this pitfall if DSLs are to be a step forward rather than backward.

Composability distinguishes jargons from conventional DSLs. Jargons are composable because their common syntax makes all jargons processable with a common interpreter, and because the interpreter is customizable with the semantics of multiple jargons simultaneously. With composable jargons, we can prevent the Balkanization of domains.

6.2. Pitfall: Cost

The cost of DSLs is a potential pitfall. A large company that embraces DSLs may end up with hundreds of DSLs instead of just a few general-purpose languages. General-purpose languages are supported by their vendors, but DSLs will most likely be supported by their users. As DSLs proliferate, their aggregate support costs can get out of hand.

Compared to a conventional DSL, a jargon is inexpensive to make, because much of the work is already done. The syntax is already designed, and the interpreter is already written. Making a jargon is reduced to designing its concrete expressions, and writing the actions for the expressions in a high-level programming language with excellent debugging facilities.

Compared to a conventional DSL, a jargon is also inexpensive to maintain. Any jargon can be extended without changing its abstract syntax, which means that the interpreter doesn't have to be changed to accommodate the extensions. Only one interpreter and one API library need be maintained for all jargons. The commonalities among jargons also help to reduce other support costs, such as for training and documentation.

The composability of jargons has ramifications for their cost and timeliness. If a problem involving multiple domains must be solved, and jargons already exist for the domains, then it may be possible to solve the problem by composing the existing jargons, thereby eliminating entirely the cost and time of making yet more jargons.

7. Related Work

Jargons are instances of little languages [Bentley86]. As such, they share the attributes of simplicity and ease-of-use that are the hallmarks of domain-specific specialization. Unlike making a jargon, making a conventional little language entails all the steps of making a "big" language: syntax design, grammar specification, parsing, and execution. These differences loom large in practice. We have found that developers who cannot make a little language can easily make a jargon in a day or two. And once made, jargons provide the advantages of composability and multipurpose models that little languages do not.

XML is closest in spirit to jargons. Languages based on XML could be used for domain engineering, but their implementations are biased toward document markup languages with their style sheets and style sheet languages for defining their semantics. This bias is not well matched to the needs of domain engineering where models are typically not documentation. Moreover, the back-end of interpreters for XML-based languages are harder to construct than for jargons. That said, the similarity between jargons and XML demands that we distinguish the InfoWiz concept from its implementation. The InfoWiz concept is founded on the following notions:

- A constant abstract syntax for all jargons
- A multiplicity of easily defined computational semantics for a given jargon
- A generic interpreter that is easily customizable with the computational semantics of multiple jargons simultaneously

The InfoWiz concept could be implemented with XML instead of WizTalk for the jargon syntax, and Perl or Java or Python instead of Fit for the host programming language. It would be good to have different implementations of the InfoWiz concept to suit different needs and tastes.

Spinellis and Guruprasad [Spinellis97] describe by way of numerous examples how software engineering is facilitated with DSLs. Despite their success, we feel that the use of conventional DSLs of the sort they advocate is ill-advised. A conventional DSL is hard to make, costly to maintain, and becomes yet another obstacle to teamwork among domains.

Faith et al. [Faith97] describe the Khepera system for making DSLs. Khepera gives the maker of a DSL the freedom to design its syntax. But our developers, who were not language experts, considered syntax design, grammar specification, and tools such as `yacc` as obstacles, not opportunities. Moreover, DSLs with different syntax are guaranteed to be incomposable. Also, the transformation approach of Khepera requires more understanding of abstract syntax trees, tree traversal and manipulation, and the inner workings of an interpreter than our developers had.

A domain-specific embedded language [Hudak96] is another approach to speed up software production. But this approach works only for languages that support rich abstraction mechanisms that make it possible to extend the base language with domain-specific extensions. Unfortunately, for reasons of legacy, our applications must be written in C. Since C does not support the abstractions needed to realize domain-specific embedded languages, we are unable to use this approach. We find such legacy constraints more the rule than the exception.

8. Lessons Learned

Jargons are up to the challenges of real-world domains. The following divide-and-conquer strategy fostered by easy-to-make jargons worked for each domain:

- Decompose the domain into subdomains
- Make a jargon for each subdomain
- Model each subdomain in its jargon
- Compose the subdomain models to model the entire domain

Although every jargon has the same abstract syntax, we found the syntax versatile enough to meet the expressive demands of their subdomains.

Domain experts can make their own jargons. In fact, we believe that domain experts make the best jargons because they have the necessary knowledge, and because they are in an environment that is conducive to rapid prototyping.

Jargons improve the quality of software. The decomposition of domains leads to higher quality software because simple models expressed in simple jargons are easier to get right. And when the simple models are composed, the resulting whole is more likely to be correct because it is built of proven components. Multipurpose models can generate multiple subproducts from a single source to ensure their consistency. Models can be translated into more comprehensible representations for review. This makes flaws in the models easier to catch. And because both the review representation and final product are generated from a single source, what is approved is what will be produced.

Jargons avoid the key pitfalls of conventional DSLs. As jargons proliferate, their aggregate cost is minimized because they are easy to make, and only one InfoWiz interpreter and API function library need be maintained for all jargons. The composability of jargons prevents the Balkanization of domains, and leverages the power of specialization through teamwork.

9. Acknowledgments

10. Availability

InfoWiz and the jargons described here are the proprietary property of Lucent Technologies. The programming language Fit is in the public domain. It is available by sending email to `lwr@research.att.com`. Contact PaceLine Technologies (`www.pacelinetech.com`) about obtaining InfoWiz.

11. References

[Bentley86] Bentley, J. Little Languages, *Communications of the ACM* **29** (8), August 1986, 711-721.

[Cuka98] Cuka, D.A. and Weiss, D.M. Engineering Domains: Executable Commands as an Example, *Proceedings 5th International Conference on Software Reuse*, Victoria, Canada, June 2-5, 1998, 26-34.

[Faith97] Faith, E.R., Nyland, L.S. and Prins, J.F. Khepera: A System for Rapid Implementation of Domain-Specific Languages, *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, CA, October 15-17, 1997, 243-255.

[Goldfarb90] Goldfarb, Charles F. *The SGML Handbook*, Clarendon Press, Oxford, England, 1990.

[Hudak96] Hudak, P. Building Domain-Specific Embedded Languages, *ACM Computing Surveys* **28** (4), December 1996.

[Johnson75] Johnson, S.C. Yacc --- Yet Another Compiler-Compiler, *Computer Science Technical Report 32*, Bell Laboratories, July 1975.

[Martersteck85] Martersteck, K.E. and Spencer, A.E. Introduction to the 5ESS(RM) Switching System, *AT&T Technical Journal* **64** (6), July-August 1985, 1305-1314.

[Nakatani97] Nakatani, L.H. and Jones, M.A. Jargons and Infocentrism, *Proceedings of DSL '97 (First ACM SIGPLAN Workshop on Domain-Specific Languages)* Paris, January 18, 1997, 59-74. Published as University of Illinois Computer Science Report, `http://www-sal.cs.uiuc.edu/~kamin/dsl`.

[Parnas76] Parnas, D. L. On the Design and Development of Program Families, *IEEE Transactions on Software Engineering* **2**, 1976, 1-9.

[Spinellis97] Spinellis, D. and Guruprasad, V. Lightweight Languages as Software Engineering Tools, *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, CA, October 15-17, 1997, 67-76.

[VanDuersen97] Van Duersen, A. and Klint, P. Little Language, Little Maintenance? *Proceedings of DSL '97 (First ACM SIGPLAN Workshop on Domain-Specific Languages)*, Paris, January 18, 1997, 109-127. Published as University of Illinois Computer Science Report, `http://www-sal.cs.uiuc.edu/~kamin/dsl`.

[XML] Anonymous. The XML Information Site, `http://www.xmlinfo.com`