♦ Automated Software Development with XML and the Java* Language

Glenn R. Bruns, Alan E. Frey, Peter A. Mataga, and Susan J. Tripp

In software development with domain-specific languages (DSLs), one defines a requirements language for an application domain and then develops a compiler to generate an implementation from a requirements document. Because DSLs and DSL compilers are expensive to develop, DSLs are seen as cost effective only when many products of the same domain will be developed. In this paper, we show how the cost of DSL design and DSL compiler development can be reduced by defining DSLs as Extensible-Markup-Language (XML) dialects and by developing DSL compilers using commercial XML tools and the Java language. This approach is illustrated through the Call View Data Language (CDL), a new DSL that generates provisioning support code and database table definitions for Lucent Technologies' 7R/E™ Network Feature Server.*

Introduction

High-level languages and compilers are a great success story of software engineering. One way of understanding this success is to see high-level languages as allowing a division of labor between problem-domain expertise, held by the application developer, and implementation expertise, held by the compiler developer. With domain-specific languages (DSLs),[1] this division is magnified. A language is created specifically for an application domain, where requirements for a product in the domain can be precisely and succinctly captured. A compiler for this language then generates the product code directly from the requirements. The developers of a DSL and its compiler are sometimes called *domain engineers*, while the writers of requirements in the DSL are called *application engineers*.

As an approach to automating software engineering, DSLs have many fascinating aspects. For example, a DSL compiler can be regarded as a formalization of the design of a family of products. Once the design is explicit and formal, the presence of design problems, inconsistencies, and complexity is apparent. Furthermore, design improvements can be implemented as changes to the DSL compiler, allowing the improvements to be applied consistently to all product instances.

DSLs are not as widely used as they might be because the return on investment for DSLs is not always clear. Designing a DSL and implementing a DSL compiler are expensive and time consuming, and they also require specialized skills. For example, just the work of analyzing an application domain prior to DSL design (called *domain analysis*) can require a team of domain experts and considerable time. Training developers to use a new language is costly. Furthermore, if the design approach for an application domain is not well known and stable, the DSL and compiler may require continual enhancement.

In this paper, we describe a lightweight approach to DSL-based development using Extensible Markup Language (XML) and the Java* language. Briefly, the idea is to define a DSL as an XML dialect and then to build the DSL compiler using the Java language and existing XML tools. While XML dialects can be verbose, they are suitable for DSLs because they are easy to define and to read. This ease of reading XML dialects stems partly from their similarity to the Hypertext Markup Language (HTML) and partly from the paucity of XML language features. As for the Java language, it is suitable for DSL compiler development because, in this task, modern language features such as object orientation and garbage collection are needed for rapid development and compiler modularity. The performance penalty one pays for these features is not an issue for DSL compilers. In short, XML and the Java language are well suited to the job of rapid DSL and DSL compiler development.

We present this approach in the context of the Call View Data Language (CDL). This language describes the data-driven customization of the services offered by the Network Feature Server (NFS), a call-processing platform that is part of Lucent Technologies' next-generation telecommunications product line. CDL has successfully been deployed as part of the NFS production process, and the generated code and schemas are in the field. The entire CDL development period, from conception to language design to compiler development, occurred within six months and was performed concurrently with the NFS development.

In the section immediately below, we provide a brief summary of the function and the services architecture of the NFS. In the "Domain Analysis" section, we define our domain analysis of the data aspects of NFS services. In the "Language Design" section, we describe XML and explain how it was used to define CDL. We follow this in the "Compiler Design and Implementation" section with a description of the architecture of the CDL compiler. In the section "Applications of CDL," we describe existing and planned applications of CDL. In the last section, we survey related work and present our conclusions.

Panel 1. Abbreviations, Acronyms, and Terms

API	application programming interface
C	high-level programming language designed at Bell Labs
C++	object-oriented descendant of C programming language, designed at Bell Labs
CDL	Call View Data Language
DOM	document object model
DSL	domain-specific language
DTD	document type definition
FAST	family-oriented abstraction, specification, and translation
HTML	Hypertext Markup Language
JSD	Jackson System Development
NFS	Network Feature Server
SAX	Simple API for XML
SGML	Standard Generalized Markup Language
SQL	Structured Query Language
XML	Extensible Markup Language
XSL	stylesheet language for XML
XSLT	XSL Transformations; language for transforming XML documents into other XML documents
`yacc`	yet another compiler compiler

Background

Lucent's 7R/E™ product family[2] provides switching solutions for voice, multimedia, and data. Elements from the 7R/E family can be combined to build customized networks that incorporate both circuit and packet transport media, switches, and endpoints. The NFS is a call-processing services element that executes the logic of the services provided to the individual or corporate customer.

An NFS call is handled by one or more communicating processes, or *call views*, as depicted in **Figure 1**. A call view represents a physical or logical entity involved in a call. For example, a call view might represent a customer or might encapsulate network provider logic (for example, the selection of a gateway to access a remote switch). The NFS architecture defines a set of call view types; each call view is an instance of such a type.

Call views have control and data aspects. The control aspect is the state machine associated with a call view's type. A data aspect exists because data is sometimes needed during execution of a call view's state machine. For example, a state machine may need data to select between two outgoing transitions at a state, one of which is taken for blocked calls and the other for unblocked calls. As another example, a state machine may need the identification of a destination switch. We call the values required by a call view the *attributes* of the call view. A call view obtains attribute values by invoking functions defined apart from the

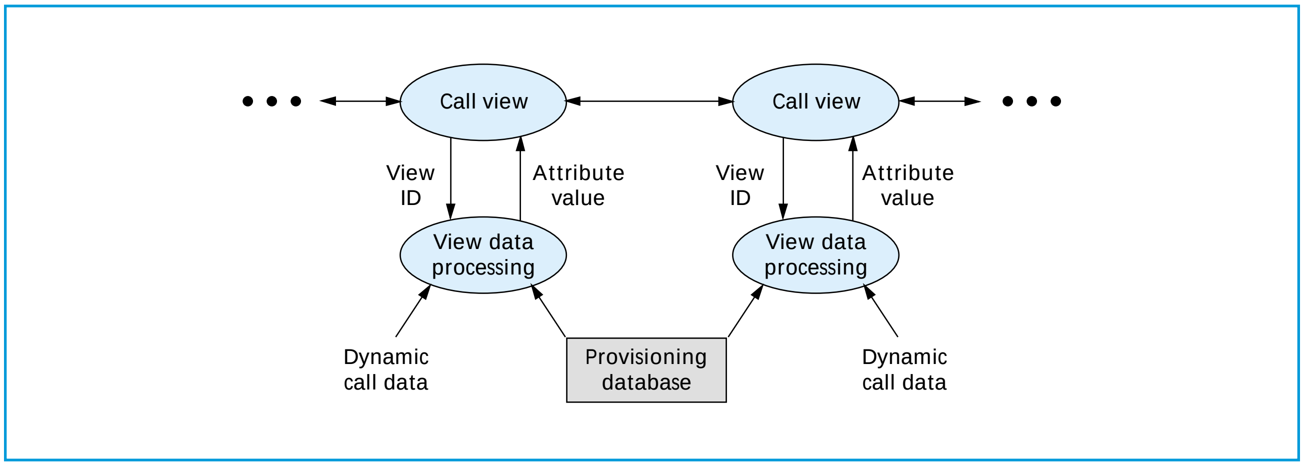

Figure 1.
NFS services architecture.

state machine. These attribute-computing functions encapsulate the data aspect of the call view and provide a way to tailor the behavior of the call view.

The functions of a call view compute their results using dynamic call parameters, such as the dialed number and the time of day, and *provisioned data*, which is data supplied by the network provider operating the system. Provisioned data can be configurational or can represent the settings of subscriber features. An example of configurational data is the list of gateways that can be used to access a particular external switch. Examples of subscriber settings are the routing policy for a company's toll-free number and the call-forwarding destination number for an individual. In the NFS, provisioned data is stored in a database.

Domain Analysis

Implementing an attribute-computing function is painstaking work. The implementation involves database design and coding at several layers of the NFS services architecture. Because the database tables and the code layers have many interdependencies, it is hard to get the pieces to fit together. Furthermore, the work is tedious, as the implementations of many attribute-computing functions are similar.

This job is clearly a candidate for automation. The first step is to analyze what attribute-computing functions are expected to have in common and how they are expected to vary. This process is called *domain analysis*.[3,4] The result of domain analysis can be used as the basis for a requirements language. In our case, the language describes the attributes of a call view.

Detailed methods exist for domain analysis. For example, the family-oriented abstraction, specification, and translation (FAST) approach[4] to developing families of software includes a domain analysis step that involves defining a decision model, establishing terminology, establishing domain commonalities, and other activities. These steps typically involve a team of domain experts working together for weeks.

With CDL, a lengthy, formal domain analysis was impossible. The need for automation within the project was only recognized after manual development was under way. The project, like most others, was under tight deadline pressure so the domain experts were not available to participate in domain analysis. Our domain analysis was, therefore, informal, relying more on existing code than on domain experts. We examined the hand-written code of various attribute-computing functions, seeing how the code could be parameterized. Finding the general pattern underlying a body of code can be difficult because of the many incidental differences between pieces of hand-written code.

Figure 2 shows the structure of our domain model in a Jackson System Development (JSD) style of notation.[5] A call view has a name and some attributes. An attribute has a name, a type, and zero or more conditions. Furthermore, an attribute can be optional, in which case the code to compute the attribute can return a special null value. A *condition* consists of a call parameter name, its type, and the kind of test that is to be made in comparing the dynamic parameter value to a static provisioned value.

Advantages of this kind of informal domain analysis are that it is fast and does not rely on the availability of domain experts. A disadvantage is that it can be done only if sample hand-written code exists. Moreover, the analysis is only as good as the sample code is representative. The approach we used lacks some of the important advantages gained when meetings of domain experts are used. For example, these meetings can be a way for the domain experts to exchange and systematize their knowledge. Additionally, bringing domain experts into the process helps assure they will support the resulting domain model.

Language Design

Designing a language is a difficult job, even if the ideas underlying the language are understood. The choices for syntax are many, and there are issues of operator precedence and associativity. Even with tools such as `yacc`,[6] the process of defining a language and debugging it to remove conflicts and unintended phrases is time consuming.

We defined CDL with XML,[7] which is a simplified descendant of the Standard Generalized Markup Language (SGML).[8] XML and SGML are *meta-languages*, through which one defines dialects. Just as HTML is an SGML dialect that captures the logical structure of hypertext documents and forms, CDL is an XML dialect that captures the logical structure of the data requirements of call views.

Roughly speaking, an XML document consists of *elements*, each of which may have *attributes* and *contained elements*. (These XML language attributes are not to be confused with NFS call view attributes.) Syntactically, the extent of an element is specified by opening and closing tags, where tags are delimited with angled brackets (<>) and specify the element type. In addition, the opening tag for an element may contain a set of attribute/value pairs, where the values must be strings. As an example XML document, see the CDL specification of **Panel 2**. Near the top of the document is a `typedef` element, which has an `id` attribute and a contained `int` element. The meaning of this specification is discussed later.

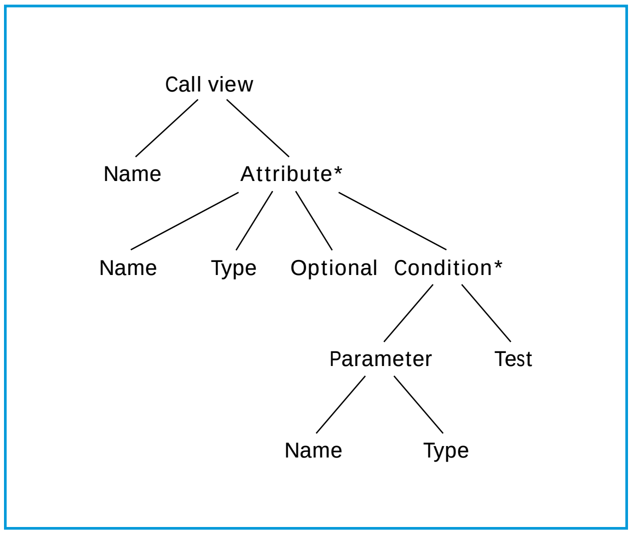

Figure 2.
A domain model for the NFS data domain.

The XML dialect designer defines the structure of a legal document as a document type definition (DTD). A DTD specifies the legal element types and their attributes. In addition, each element has a content model (a simple grammar) that specifies how elements can be nested within it. This allows limited, but useful, syntactic validity checking of XML documents. Any additional checks are outside the scope of the DTD.

Because of XML's simplicity, an XML dialect designer does not face many of the issues usually met in language design. There are virtually no decisions that need to be made about syntax. Operator precedence and associativity issues are not relevant. The main decision that is faced is whether information associated with an element should be captured as a contained element or as an XML attribute.

Panel 3 is a simplified version of the DTD for CDL. The work to develop the DTD from the informal domain description of Figure 2 was straightforward, taking us only about two hours despite having no XML experience. Reading through the DTD, we see that element `cdl` contains type definition elements, parameter elements, attribute elements, and view ele-

Panel 2. An Example CDL Specification

```
<cdl>
<typedef id="lineTypeValue">
  <int low="0" high="99" />
</typedef>

<typedef id="ScreenValue">
  <desc> Whether to block a call. </desc>
    <enum>
      <token short-name="Y" long-name="Block"/>
      <token short-name="N" long-name="DontBlock"/>
    </enum
</typedef>

<parameter id="lineType" type="lineTypeValue"/>

<attr id="Screen" type="ScreenValue">
  <condition param="lineType" test="equal" />
</attr>

<view id="Switch">
  <view-attr attr="Screen"/>
</view>
</cdl>
```

ments, in any order. A `view` element has an `id` attribute, and contains an optional description element followed by zero or more `view-attr` elements. A `view` attribute has a single required attribute `attr`. It does not directly contain attribute elements because we expect attributes to be shared between views. An `attr` element has `id`, `type`, and `optional` attributes. The `id` and `type` attributes are optional, while the `optional` attribute has default value "`no`." The remainder of the DTD should be understandable from the parts we have just paraphrased.

The minimality of XML leads to some clumsiness in CDL specifications and limits the checking that can be performed by an XML parser. For example, referring again to Panel 2, we would have liked to say that `low` and `high` attributes must be given integer values. Instead, like all XML attributes, they take string values. The string quotations are awkward, and an XML parser does not check that the given strings represent legal integer values. As another example, consider the `attr` attribute of the `view-attr` element in this panel. We would have liked to say that the value of the `attr` attribute must be the `id` of an `attr` element in the specification. The best one can do in XML is to define the `attr` attribute to be an `IDREF`, which means that its value must be the `id` of some other element in the specification.

We have said nothing yet about what a CDL specification means. Informally, it simply declares types—a set of call view types and the types of the attribute-computing functions associated with each call view. In Panel 2, a single `Switch` call view type is declared. It has only attribute `Screen`. The function to compute attribute `Screen` maps a value of type `lineTypeValue` (the type of `Screen`'s parameter) to a value of type `ScreenValue` (the type given for `Screen` itself). An attribute may be defined to have zero parameters, in which case the function is understood as a constant, or may have multiple parameters, in which case the function has multiple input arguments. If `Screen` were optional, the function could produce either a `ScreenValue` as output or a special `null` value.

Since a CDL specification declares only types, not

Panel 3. CDL Language Definition

```
<!ELEMENT cdl (typedef | parameter | attr | view)*>

<!ELEMENT parameter (desc?)>
<!ATTLIST parameter
  id       ID     #REQUIRED
  type     IDREF  #REQUIRED>

<!ELEMENT desc (#PCDATA)>

<!ELEMENT attr (desc?, (condition)*)>
<!ATTLIST attr
  id       ID     #REQUIRED
  type     IDREF  #REQUIRED
  optional (yes | no) "no">

<!ELEMENT condition EMPTY>
<!ATTLIST condition
  test     (equal|not-equal|less-than|greater-than) "equal"
  param    IDREF  #REQUIRED>

<!ELEMENT view (desc?, (view-attr)*)>
<!ATTLIST view
  id       ID #REQUIRED>

<!ELEMENT view-attr EMPTY>
<!ATTLIST view-attr
  attr     IDREF #REQUIRED>

<!ELEMENT typedef (int | enum | record)>
<!ATTLIST typedef
  id       ID #REQUIRED>

<!ELEMENT int    EMPTY>
<!ATTLIST int
  low      NMTOKEN "0"
  high     NMTOKEN "1000000">

<!ELEMENT enum (token)+>
<!ATTLIST enum
  base-type (int|char) "char">

<!ELEMENT token (desc?)>
<!ATTLIST token
  short-name NMTOKEN #REQUIRED
  long-name  ID      #REQUIRED>

<!ELEMENT record (field)+>

<!ELEMENT field (int | enum)>
<!ATTLIST field
  id       ID #REQUIRED>
```

operations, one may wonder how the CDL compiler can produce running code. The explanation is that the attribute-computing methods it generates are data driven, so that the value returned by a method depends on the contents of database tables associated with the call views and attributes. One can think of the generated methods as functions that map call parameters and database table values to attribute values. Thus, the code generated by the CDL compiler is part of the mechanism by which the behavior of the NFS (such as how calls are routed) is customized using the values in the NFS provisioning tables.

The simple CDL specification of Panel 2 fails to illustrate many CDL features. For example, a CDL specification may contain declarations of multiple call view types, two or more call view types may refer to the same attribute, and an attribute may have multiple conditions. The CDL specification used in the NFS is about 2800 lines long, and contains declarations of about 50 views and 300 attributes.

Compiler Design and Implementation

The main job of the CDL compiler is to generate the code and database tables needed by the attribute-computing functions of a call view. In fact, the NFS services architecture has three layers for which the CDL compiler must produce outputs. As shown in Figure 1, the provisioning database is the bottom layer. The "view data processing" layer of the figure actually represents two layers of the services architecture. The lower of these two is an interface layer that insulates peculiarities of the database interface from higher levels and provides a transaction batching facility. The higher of the two, called the *data reader*, is responsible for reading call parameters, reading values from database tables, and making computations. Some of the code in these layers is hand-written infrastructure code that does not change; the CDL compiler generates only the code that changes depending on the particular attributes of the call views.

Besides generating C++ code, "include" files, and Structured-Query-Language (SQL) table definitions, the compiler generates HTML data design documentation. Two kinds of documentation are produced. The first is a set of table descriptions that shows the structure of each CDL-related database table, as well as data integrity constraints associated with the table. The second is a data dictionary that shows the name, type, and data domain for each field of each table.

Figure 3 shows the phases of the CDL compiler: parsing and error checking, transformation, and code generation. For the parsing phase, we used IBM's XML parser,[9] one of the many publicly available XML parsers. This parser takes an XML document as input, validates it against the DTD, and produces as output a Java object that uses the document object model (DOM)[10] application programming interface (API). The DOM provides a standard interface for Java programs to use in constructing, manipulating, and examining XML documents. Optionally, the IBM parser can produce Java objects that use the Simple API for XML (SAX),[11] an API for event-based XML parsing.

In early versions of the CDL compiler, the subsequent compiler phases accessed specification-related information directly through the DOM. A problem with this approach is that changes to CDL syntax lead to changes in many parts of the compiler. Another problem is that checks for CDL language errors not expressible in the DTD are made each time the DOM is used. Furthermore, the DOM interface is sometimes awkward. For example, to access an attribute of a CDL call view, one must find the value of the `view-attr` attribute and then find the `attribute` element referenced by it.

These problems led us to define a CDL-specific API containing classes such as `View`, `Attr`, and `Param`. Now, as a second parsing step, the CDL compiler accesses the DOM to create CDL-related objects. This phase also contains the many CDL-related type checks that cannot be expressed in a DTD. The DOM is accessed by the compiler only in this second parsing step.

The main work of the compiler is in the transformation phase, which generates intermediate and output-related objects from the specification-related objects. The main output classes correspond to the various layers of the service architecture. For example, the compiler has a `DataReader` class, an `ADM` class, and a `DB` class. The `DataReader` and `ADM` classes contain a Java method for each kind of C++ method to

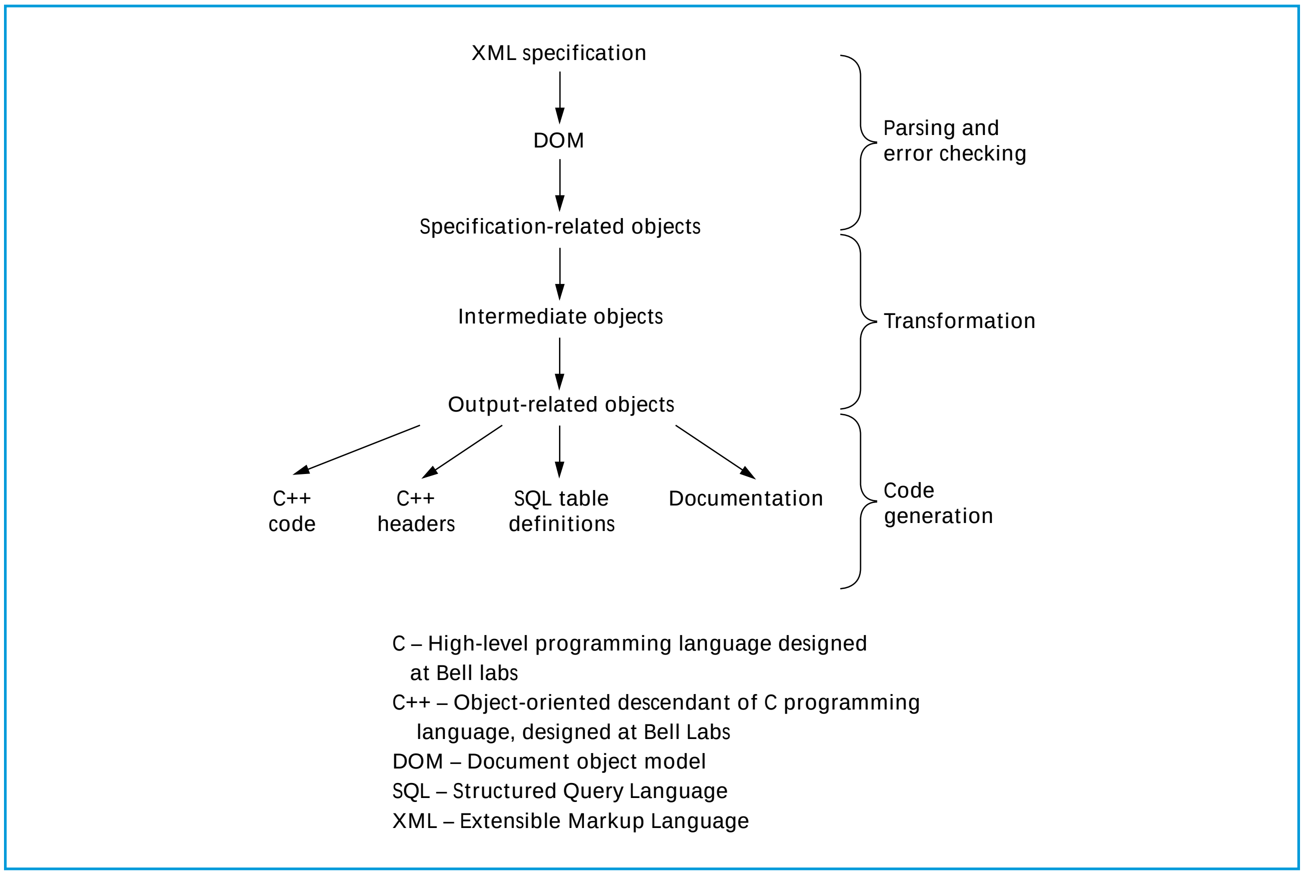
XML specification
DOM
Specification-related objects
Intermediate objects
Output-related objects
C++ code
C++ headers
SQL table definitions
Documentation
Parsing and error checking
Transformation
Code generation
C – High-level programming language designed at Bell labs
C++ – Object-oriented descendant of C programming language, designed at Bell Labs
DOM – Document object model
SQL – Structured Query Language
XML – Extensible Markup Language

Figure 3.
Phases of the CDL compiler.

be generated as output in these layers. The `DB` class contains a method for each kind of database table that can be generated. Intuitively, the values of the specification-related objects serve as parameters to the constructors of the output objects. In some cases, the parameterization is simple, such as the case where the name of a database table is derived from the name of a view in the CDL specification. In other cases, the parameterization is complicated. For example, attribute-computing functions in the generated data reader code vary greatly in structure and size, depending on many parts of the CDL specification.

The code generation phase is simple. Print methods of the intermediate and output-related classes generate code, table definitions, and documentation.

The structure of the compiler changed significantly during development. The main improvement resulted from applying the principle that code dealing with syntax of the source or target language should be isolated from the code dealing with object transformations. As an example of how this principle was applied, consider the generation of database-related outputs. For each SQL table definition that is generated, a similar C++ structure declaration must be generated. In an early version of the compiler, a single method existed for each type of table to be generated. The method contained print statements to output the SQL definitions and the C++ declarations. The code was a tangle of Java statements and many literal strings representing fragments of C++ and SQL. When the method was changed, care was needed to keep the SQL generation and C++ generation consistent. This code was improved by creating a new `Table` class representing the abstract structure of a database table. The CDL compiler now constructs a `Table` object; then during code generation, the object is printed as an SQL table definition and as a C++ structure declaration. Similarly, `Method` and `Class` classes were defined to capture the abstract structure of C++ methods and tables.

We built the CDL compiler after defining the CDL language, but were we to start a similar project today, we would do things differently. From the domain analysis, we would define the abstract structure of the language, possibly as Java classes. Next, we would define an XML dialect and write the parser phase of the compiler. We would then define Java classes for the output classes and write the print methods for them. Finally, we would find appropriate intermediate classes and write the transformation phase of the compiler.

The circumstances under which we developed the CDL compiler were peculiar but perhaps likely to be faced by others who attempt DSL-based development. When we first recognized the potential benefit of a DSL for the project, NFS software development was already under way. Project deadlines were too tight to permit the developers to build a CDL compiler or even to help us in doing so. Therefore, having little knowledge of the NFS software architecture, we worked by examining partial and preliminary code. Our work had a strong reengineering aspect, as much of our time was spent looking for general patterns in the hand-written code.

Many of the problems we faced in developing the compiler were related to this style of working. In particular, it was often required that generated artifacts conform to existing hand-written artifacts. For example, this was the case for database table definitions. The definitions were fixed because they had been standardized previously and shared with an organization working on a related NFS provisioning subsystem. The tables contained various optimizations in which fields were combined to save space. A major challenge in writing the compiler was to find general rules for table generation that would produce tables with exactly the same fields as those produced by the human designer. With much work, we found a rule that could duplicate the hand-applied optimizations. Compatibility of generated outputs with hand-produced outputs was a critical factor in the acceptance of CDL.

Applications of CDL

The first release of the NFS software shipped with the platform in the fourth quarter of 1999. The CDL compiler is now maintained by the NFS development organization, and the CDL specifications and generated code are under version control. In collaboration with the development team, we have since extended CDL and the CDL compiler to allow more code and database tables to be generated. This extension nearly doubled the amount of CDL-generated output. Currently about 30,000 lines of C++ code and SQL are generated from a CDL specification of about 2800 lines. (In both cases, these figures include blank lines and comments.) However, the real advantage of using CDL is *not* that less CDL code needs to be written than C++ and SQL code; it is that the CDL is much easier to write. Adding support for a new attribute in an NFS call view requires adding a block of about ten lines in the CDL specification. Making the corresponding modification by hand would require changing five to ten separate source, header, SQL, and documentation files having different owners and written in different languages.

In the initial release of the NFS, CDL-generated code accounted for only a fraction of the overall platform code. However, since much of the platform code is for infrastructure, the CDL contribution has increased as the NFS feature set has grown, and CDL has a significant role in speeding incremental feature development.

There are many further opportunities for automation from CDL specifications. For example, the CDL compiler could generate the Web pages and database code used in the data provisioning system used with NFS. It could generate a similar data provisioning system used by NFS system testers. It could also generate code to enforce the integrity constraints required of the provisioned data.

Related Work and Conclusions

The important issue for DSLs and code generation is not whether they work, but how they can be developed cheaply and without specialized expertise in domain analysis, language design, and compiler development. In our work on CDL, the compiler development was by far the most costly part. As described in the section “Compiler Design and Implementation,” the code generation part of the compiler includes many target language fragments, making it difficult to read the compiler code, easy to introduce syntax errors in generated code, and difficult to port the compiler to different target languages and architectures.

A code generation technique that solves some of these problems is *language embedding*. Here, expressions in the target language are represented as data structures in the compiler. Liejen and Meijer[12] use this technique for SQL generation. They show that target language data types can be defined in Haskell so that if the compiler itself type checks, then so will the code it generates. However, language embedding is cumbersome when the target language is a general-purpose programming language. A multitude of data types must be defined for all the language constructs, and representing simple expressions of the language requires complicated data structures. In our CDL compiler, language embedding was only used to the extent that the compiler has Java classes for C++ methods and classes. In these classes, the body of a method is represented as an unstructured character string.

Template languages can also help in code generation. Here, the target language is extended so that programs with "holes" can be defined; one instantiates such templates at compile time by filling the holes with expressions of the compiler language that evaluate to expressions in the target language. A C++ template is a weak form of template in which only type parameters can be used. A template language for HTML is described by Ladd and Ramming.[13] Unfortunately, the degree of parameterization needed for many CDL compiler outputs is too large to be captured in the template style.

The CDL compiler could also have been defined using a language transformation system such as XSLT.[14] In this system, the code of an XML dialect is transformed to HTML, another XML dialect, or some other language through a sequence of applications of transformation rules. We experimented with XSLT but found the transformations hard to maintain. The difficulty was that the compilation requires the gathering of information declared in a variety of places within the CDL document, and a purely transformational approach tends to require duplicated side processing that is hard to understand.

The InfoWiz™ system[15] supports a template style but also allows rich internal data structures. Code generation with XML and the Java language has strong parallels to the InfoWiz approach. In both approaches, DSLs are defined as dialects of a simple base language, making language design easy. The InfoWiz analog of XML is WizTalk; a WizTalk dialect is called a jargon. In both approaches, a modern language is used to implement the code generation. The InfoWiz analog of the Java language is FIT. A difference in the approaches is that a fixed scheme for language interpretation is used in InfoWiz. A language implementer using InfoWiz defines an action for each special term of a WizTalk dialect; a generic interpreter then traverses the parse tree of a WizTalk program in depth-first, left-to-right order, applying the appropriate action to each node of the tree. This approach eliminates some of the work of compiler implementation and makes it easy to combine jargons.

The relative advantages of XML and the Java language versus InfoWiz depend heavily on circumstances. In our application, a simple, single DSL was used but the generated code was intricate, and code generation used intermediate structures. In other applications, in which the code generation is reasonably simple and multiple DSLs are used together, InfoWiz may be a better choice. A drawback of InfoWiz is that it does not follow some important technology trends. XML does roughly the same job as WizTalk, but only XML has a large and growing body of tool support. The Java language and FIT are also similar, but only the Java language is widely known, standardized, and supported. These technology trends affect not only short-term project costs—they also matter because developers and their managers see value in building expertise in emerging technologies.

We have argued here that XML and the Java language work together well in the lightweight development of DSLs and DSL compilers. Interestingly, the claim that XML and the Java language work together well in general[16,17] is based on a different argument. The argument for the general case is about portability: XML supports portable data, while the Java language supports portable programs. The argument for our case is about rapid development: XML supports the rapid development of simple, familiar languages, while the Java language supports rapid compiler development through its modern programming constructs. We also cite the availability of XML tools based on the Java language as a factor for rapid compiler development; these tools exist perhaps because of the general synergy between the languages.

Acknowledgments

We would like to acknowledge the contributions of the NFS development team to CDL, in particular Moh Barathan, Yumei Dang, and Joanna Wang of Lucent's Switching and Access Solutions Group. We also had fruitful discussions with Michael Benedikt of Bell Labs' Software Production Research Department.

***Trademark**
Java is a trademark of Sun Microsystems, Inc.

References

1. T. J. Ball, ed., *Proc. of the Second USENIX Conf. on Domain-Specific Languages*, Austin, Tex., Oct. 3–5, 1999.
2. Lucent Technologies Switching and Access Solutions, *7R/E™ Packet Solutions*, <http://www.lucent-sas.com/7re>.
3. D. L. Parnas, "On the Design and Development of Program Families," *IEEE Trans. on Software Engineering*, Vol. Se-2, No. 1, 1976, pp. 1–9.
4 D. M. Weiss and C. T. R. Lai, S*oftware Product-Line Engineering: A Family-Based Software Development Process*, Addison-Wesley, Reading, Mass., 1999.
5. M. A. Jackson, *System Development*, Prentice Hall, Upper Saddle River, N.J., 1983.
6. B. W. Kernighan and R. Pike, *The UNIX Programming Environment*, Prentice-Hall, Upper Saddle River, N.J., 1984.
7. T. Bray, J. Paoli, and C. M. Sperberg-McQueen, ed., "Extensible Markup Language (XML) 1.0," REC-xml-19980210, W3C, Feb. 10, 1998, <http://www.w3.org/TR/REC-xml>.
8. C. F. Goldfarb, *The SGML Handbook*, edited by Y. Rubinsky, Oxford University Press, New York, 1990.
9. IBM alphaWorks, *XML Parser for Java*, <http://www.alphaworks.ibm.com/formula/xml>.
10. V. Apparao, S. Byrne, M. Champion, S. Isaacs, A. Le Hors, G. Nicol, J. Robie, P. Sharpe, B. Smith, J. Sorensen, R. Sutor, R. Whitmer, C. Wilson, "Document Object Model (DOM) Level 1 Specification," Ver. 1, edited by V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood, REC-DOM-Level-1-19981001, W3C, Oct. 1, 1998, <http://www.w3.org/TR/REC-DOM-Level-1>.
11. Megginson Technologies, *SAX 2.0: The Simple API for XML*, <http://www.megginson.com/SAX>.
12. D. Leijen and E. Meijer, "Domain Specific Embedded Compilers," *SIGPLAN Notices Conf. Proc.*, Vol. 35, No. 1, ACM, Jan. 2000, pp. 109–122.
13. D. A. Ladd and J. C. Ramming, "Programming the Web: An Application-Oriented Language for Hypermedia Service Programming," *Proc. of the 4th Intl. World Wide Web Conf.*, Boston, Mass., Dec. 11–14, 1995, <http://www.w3journal.com/1/h.251/paper/251.html>.
14. J. Clark, ed., "XSL Transformations (XSLT)," REC-xslt-19991116, W3C, Nov. 16, 1999, <http://www.w3.org/TR/xslt>.
15. L. H. Nakatani and M.A. Jones, "Jargons and Infocentrism," *Proc. of First ACM SIGPLAN Workshop on Domain-Specific Languages*, ACM, Jan. 18, 1997, pp. 59–74.
16. J. Bosak, "XML, Java, and the Future of the Web," Mar. 10, 1997, <http://metalab.unc.edu/pub/sun-info/standards/xml/why/xmlapps.htm>.
17. K. Sall, "XML and Java: Why These Two," Nov. 16, 1998, <http://wdvl.com/Authoring/Languages/XML/Java/perfect_pair.html>.

(Manuscript approved July 2000)

GLENN R. BRUNS is a member of technical staff in the Software Production Research Department of Bell Labs in Naperville, Illinois. He received a B.S. in chemical engineering from California State University in Northridge, an M.S.E. from the Wang Institute in Tyngsboro, Massachusetts, and a Ph.D. in computer science from the University of Edinburgh in Scotland. Dr. Bruns is involved in the development of languages, theory, and tools for the automated production and analysis of networking software.

ALAN E. FREY is a Bell Labs Fellow and a consulting member of technical staff in the 7R/E™ Multi-Services Feature Development Department of Lucent Technologies' Service Provider Networks Group in Naperville, Illinois. He has both a B.S. and an M.S. in electrical engineering from Rice University in Houston, Texas. Currently, he is working on the call processing architecture and design for the 7R/E Packet Toll/Tandem Solution.

PETER A. MATAGA is a former member of technical

staff from the Software Production Research Department of Bell Labs in Naperville, Illinois. He has a B.Sc. in mathematics and a B.E. in engineering science from Auckland University in New Zealand as well as an S.M. in engineering and a Ph.D. in engineering sciences from Harvard University in Cambridge, Massachusetts. A recipient of the National Science Foundation's Presidential Young Investigator Award, he focused his research at Bell Labs on service creation and domain-specific languages, especially for converged Web and telephony services.

SUSAN J. TRIPP is a distinguished member of technical

staff in the 7R/E™ Multi-Service Development Department of Lucent Technologies' Switching and Access Solutions Group in Naperville, Illinois. She received a B.S. in mechanical engineering from the University of Illinois in Champaign and an M.S. in computer science from Northwestern University in Evanston, Illinois. Her responsibilities include development of the call processing architecture and services for the 7R/E Packet Tandem Solution. ◆