TOOLS FOR BUILDING APPLICATION GENERATORS

J. Craig Cleaveland and Chandra M. R. Kintala

One approach to improving software productivity is to reuse as much existing software as possible. However, straightforward reuse of software is often inappropriate and sometimes inefficient. Application generators (i.e., program generators suitable for a particular application domain) can be used to produce customized code from existing software automatically without any loss in the efficiency of generated code. This paper describes three tools for building application generators: *Stage*, *PG2*, and *WOODS*. In creating application generators, these tools generate underlying translation routines automatically, allowing a programmer to concentrate on the target application domain and not on translation and compiling.

J. Craig Cleaveland *is a member of technical staff in the Software Development Technology Department of AT&T Bell Laboratories in North Andover, Massachusetts. Mr. Cleaveland's research is in the area of software development tools. He joined the company in 1982 and has a B.A., an M.S., and a Ph.D. in computer science, all from the University of California–Los Angeles.*

Chandra M. R. Kintala *is a supervisor in the Advanced Software Department of Bell Laboratories in Murray Hill, New Jersey. Mr. Kintala is responsible for research on advanced programming environments. He joined AT&T in 1980 and has a B.Sc. in electrical engineering from Regional Engineering College, Rourkela, India; an M.Tech. in electrical engineering from the Indian Institute of Technology, Kanpur, India; and a Ph.D. in* (continued on page 58)

Introduction

A simple form of software reuse is through subroutine libraries, macro languages, and, more recently, through object libraries in object-oriented systems. Such reuse of software is only feasible in limited parts of a large application system, such as a telecommunications system. However, any organization has a great deal of other software (new and old) that could be reused after some customizing.

For example, in a communications subsystem to connect two other systems developed in AT&T, many translation routines were needed for specialized messages. These translation routines were similar but not the same; thus, a straightforward reuse would have been inappropriate. Manual customization (i.e., programmers keeping informal notes on reusable software and "cutting and splicing" pieces of software to produce new code) has several drawbacks.[1]

- Programmers do not have any systematic or standard way to describe just what portions of the reusable software are relevant and how they need to be changed.
- Customizing is time-consuming, tedious, and error-prone.
- Once the process is finished, both the old and new software must be maintained as if each were unique.

One solution is to provide for automatic customization of reusable software. The Draco approach for automatic customization from reusable components is based on a series of domain-specific program transformations.[2] Our approach is to use application generators. Application generators automatically produce customized code from an existing base of software without any loss in the efficiency of generated code. Such generators have a set of generic software modules with formal (i.e., machine processable) notations for customization in a given instance. This type of customization offers more predictable and productive reuse of software and produces more reliable and efficient code. Customization can be tightly controlled, thus providing a method for enforcing standards.

If a generator for a given application is already available, it is better to use it directly. Indeed, application generators have been used successfully in many areas such as data processing, databases, and user interfaces.[3] New approaches for generating data-processing applications continue to be popular.[4] However, most application generators, such as commercial report generators, are not generally useful in large and specialized communications and real-time systems software. Thus, one is often faced with the problem of building an application generator for a limited use within one's own system.

Building an application generator is, to say the least, difficult. It requires:

- Skill in translator/compiler writing
- Knowledge of the target application domain and the ability to build generic units of reliable software in that domain
- Skill in designing specification languages and user interfaces for the application generator.[5]

Moreover, the need for additional development resources to build application generators for limited use within a project may sometimes prolong the development effort. This additional development cost can, however, be amortized over several applications.

Bassett advocates building program generators for automatic customization of reusable software; however, his program generators have only four meta-operations for cutting and splicing programs.[1] This paper describes three tools developed by AT&T Bell Laboratories for building application generators semiautomatically. They are Stage, PG2, and WOODS. SSAGS (for "syntax and semantics analysis generation system") is another of these tools.[6] Such tools are known as *application generator generators.*

Panel 1. Specification for the Word Sets Example

```
colors { red, blue, green }

cities { boston, newyork }

bugs { ant, spider, fly, moth, bee }
```

Building an application generator in this way requires only a knowledge of the application domain. The tools automatically produce the other components (translator, application code generator, and user interface and language functions of the application generator).

Concepts and Technology

When building an application generator, we want to identify groups of applications with a common "structure" in their specifications for which there is a common "structure" in the programs to implement those applications. If the specifications can be formalized and the program structures generalized as annotated program abstractions, a generic code generator can interpret the program abstractions against a specification to generate the target program.

The specifications can be a language, an interactive dialogue, a diagram, or an *attributed-tree* data structure. (When languages are used, they are sometimes referred to as fourth-generation languages, application-oriented languages, little languages,[7] or simply specification languages.) The output can be a code segment, a program, or an entire application.

Applications and Program Abstractions. Suppose there is a need to classify a collection of about 1000 words into various named sets of words within a program. Panel 1 shows a sample of this word collection in a natural notation. This expresses *what* the collection is. We will use this Word Sets example throughout the paper. Panel 2 shows how this collection is implemented using the C program-

Panel 2. A Program for the Word Sets Example

```
int number_of_sets = 3;

char *name_of_set[] = {
     "colors",
     "cities",
     "bugs", };
int size_of_set[] = { 3, 2, 5, };

char *set_of_colors[] = {
     "red",
     "blue",
     "green", };

char *set_of_cities[] = {
     "boston",
     "newyork", };
char *set_of_bugs[] = {
     "ant",
     "spider",
     "fly",
     "moth",
     "bee", };

char **values_of_set[] = {
     set_of_colors,
     set_of_cities,
     set_of_bugs, };
```

ming language. Each set of words is represented by an array of strings. The collection of word sets is represented by arrays that give the name, size, and values of each set.

It is much easier to read, write, and change the specification of this collection than to change program code. Traditionally, a programmer would take the specifications and manually generate the program fragment in Panel 2. However, this manual process is labor-intensive both initially and when making future changes or writing a program for a slightly different specification. It is also prone to error. Sizes must be kept consistent, and the `size_of_set`, and `values_of_set` must be consize_of_set, and `values_of_set` must be consistent. For example, consider the changes needed to delete the `cities` set:

1. The value of `number_of_sets` must be decreased by 1.
2. The name of the set must be removed from `name_of_set`.
3. The size must be removed from `size_of_set`.
4. The array `set_of_cities` must be removed.
   `values_of_set`.
   `values_of_set`.
6. The `cities` set must be removed from the specification.

Using macros and/or subroutines in the original program will not help modify the program. Any good programmer who writes and maintains these kinds of programs usually keeps a copy, such as the one shown in Panel 2, along with some annotations describing how to get another program for a different specification in that application domain. We call a program with such annotations a

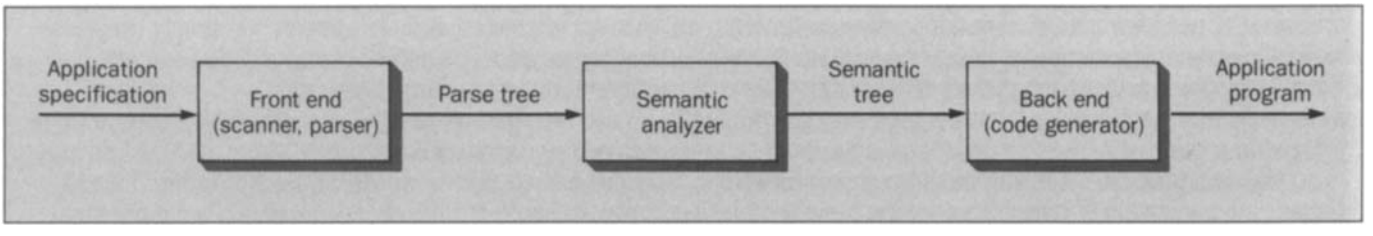

Figure 1. Architecture of an application generator.

Panel 3. An Informal Program Abstraction for the Word Sets Example

```
int number_of_sets = number of sets in the specification;

char *name_of_set[] = {
for each set name n, loop:
     "name of n",
end-loop };

int size_of_set[] = {
for each set s, loop: number of elements in s, end-loop };

for each set s, loop:

char *set_of_name of s[] = {
     for each element e in set s, loop:
          "name of e",
     end-loop };
end-loop

char **values_of_set[] = {
for each set name n, loop:
     set_of_name of n,
end-loop };
```


program abstraction. Panel 3 is a program abstraction with informal annotations for our example.

Application Generators. If there are formal (i.e., machine processable) annotations in the program, an application generator can be built that takes as input a specification of the form in Panel 1 and produces a program of the type in Panel 2 by interpreting the formal counterpart to the program abstraction in Panel 3 against that specification.

If such an application generator exists, it is preferable to use that. As previously stated, however, these kinds of specialized application generators are seldom available off the shelf. If one decides to build such a generator, it is important first to understand its architecture.

Broadly speaking, an application generator has three modules. (See Figure 1.) The front end of the application generator scans the input specification, checks for its syntactic correctness, and produces an intermediate data structure (often in the form of a tree, known as a *parse tree*). Figure 2 shows a possible parse tree for our Word Sets example.

The semantic analyzer processes the parse tree and embeds semantic information into the tree, often through attributes in the individual nodes. The names of the nodes in the parse tree in Figure 2 are the values of the name attributes of those nodes. An *attribute* is a value associated with the nodes of a parse tree. Attributes are used for computing semantics of languages, type checking, and translation.

The back end of the generator traverses the parse tree and interprets it to produce the equivalent program. Information about how to traverse the tree and how to

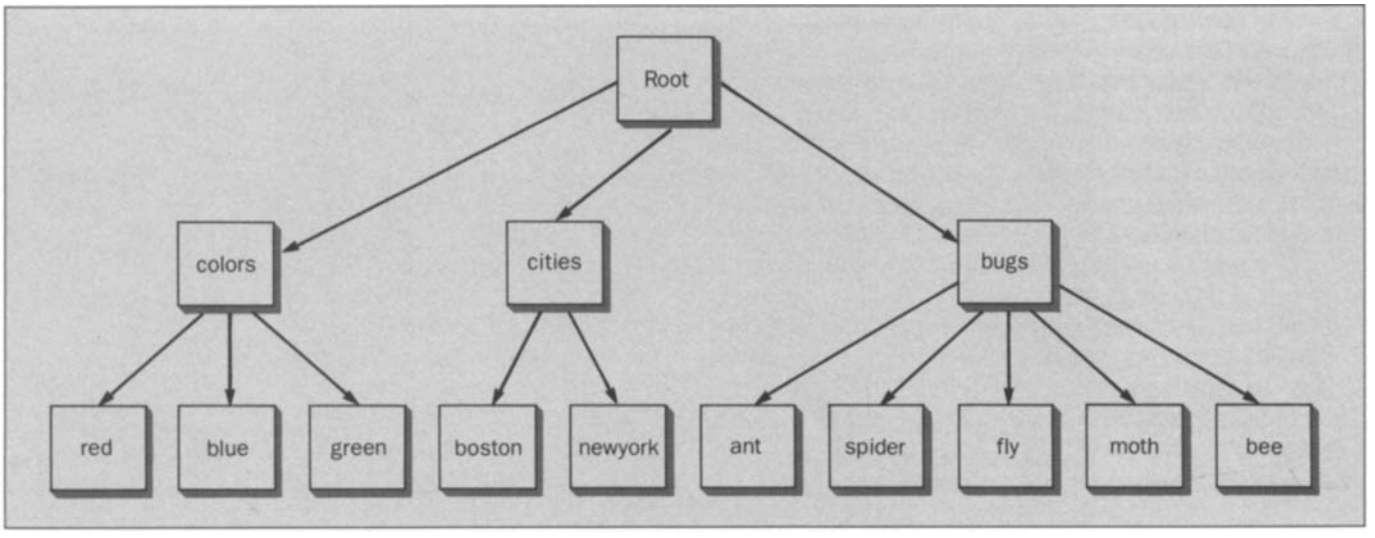

Figure 2. A parse tree for the Word Sets example.

interpret it is embedded either within the application generator or in a separate file such as a program abstraction. For example, the program abstraction in Panel 3 requires a way to traverse the parse tree from the root to the next level, obtain the name attributes of the second-level nodes, and substitute those name strings in the line specified.

This method of translating specifications into programs is similar to, albeit simpler than, the process of compiling because application languages tend to be much less complex in their syntax and semantic models than programming languages. In addition, target programs are in a high-level programming language, unlike the assembly languages of compilers. This method of generating code by interpreting and making structural expansions of the constructs in the program abstraction based on an intermediate semantic tree is analogous to a combination of the interpretive and table-driven methods used in retargetable code generators for compilers.[8,9] This translation method has also been viewed as a tree transformation method.[10]

Aside from the similarity to compilation, application generators based on program abstraction have several advantages in practice. They offer more flexibility in the sense that data structures and program structures are more easily altered. For example, changing from an array implementation to a linked-list implementation by changing the program abstraction is much easier than changing generated code manually. In addition, a different target application program can be generated from the same input specification by simply plugging in a different program abstraction. This is useful for generating other products such as standardized documentation or test scripts.

Application Generator Generators. Building application generators of the type discussed in the previous section requires a knowledge of translator/compiler writing as well as a knowledge of the target application domain. Stage, PG2, and WOODS are tools that will generate underlying translation routines *automatically*, thus allowing the builder of the application generator to concentrate on the target application domain.

As shown in Figure 3, the application generator is built from a predetermined mix of generated code (consisting of routines and data) and generic library code. The actual mix depends on the application generator generator used. The specification of the application generator (consisting of a description of the application specification language and the program abstraction) can be "compiled" or "interpreted." Figure 3 assumes that the application generator specification is compiled to generate the applica-

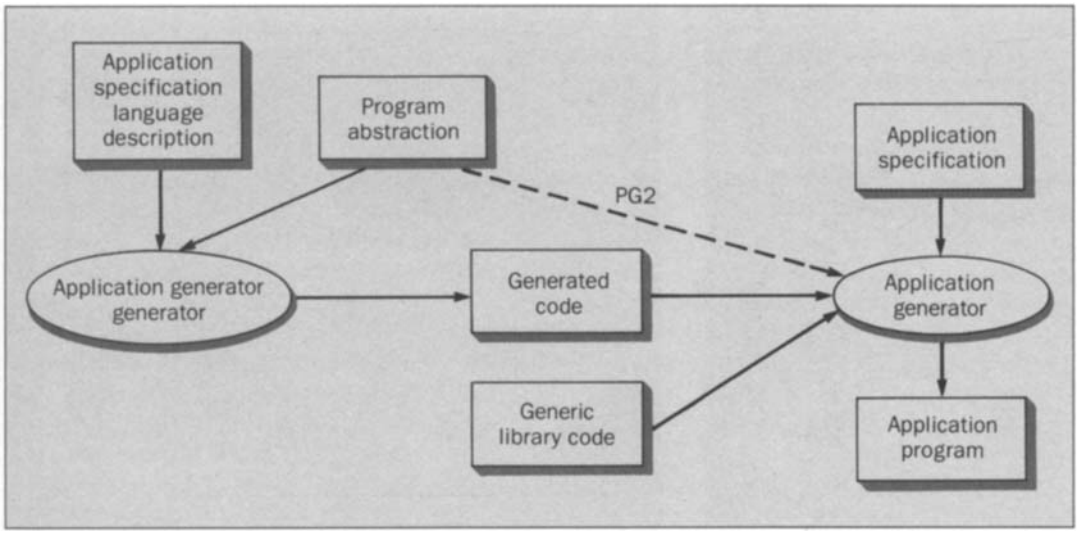

Figure 3. Architecture of an application generator generator.

tion generator. The dashed line shows that PG2 interprets the program abstraction.

The front end of the constructed application generator is generated from a description of the application specification language. This type of generator is often similar to a parser generator such as `yacc` (for "yet another compiler-compiler") on the UNIX® system. The description contains language syntax, attribute declarations, and domain-specific attribute computations. This part of the application generator also includes data structures for the tree nodes, and routines to traverse the tree or access information from tree nodes. The back end of the application generator is described with a program abstraction that expresses—in a formal language— traversals, controlled structural expansions of code fragments, and manipulation and use of semantic information in the semantic tree (as in Panel 3).

Although Stage, PG2, and WOODS share this general architecture, they differ from each other in varying degrees of detail. Stage provides all the parts and modules discussed in this section. PG2 *interprets* rather than compiles the program abstraction. In addition, PG2 expects the application generator builder to provide the front end and the semantic analyzer, thus providing flexibility (at the cost of developing those two parts) in the input mechanisms, i.e., forms, menus, or language-based interactions. While Stage builds a language-oriented application generator, WOODS builds a dialogue- or menu-oriented application generator.

Stage

The input to Stage consists of two parts:

- A *source description* describing the application specification language
- One or more *product descriptions* (such as program abstractions) describing the output of the application generator. (See Panel 4.)

From these specifications, Stage produces an application generator. We will use our Word Sets example to show how Stage works.

Panel 4 shows a file (`mktab.sd`) containing the source description (hence the `.sd` suffix) for an application generator named `mktab`. (See Figure 4.) The source description is a grammar for the specification language. The first two definitions are grammar rules. The first, the start rule, defines a `spec` as a sequence of `set_defs`. The second rule states:

- A `set_def` is a `set_name` followed by a `"{"`, followed by a list of one or more `elements`, separated by commas and finally ending with a `"}"`.

Panel 4. Source Description File: `mktab.sd`

```
%grammar

spec:     ( set_def+ )

set_def:  ( set_name "{"
            element+ separated-by ","
            "}"
          )

set_name: <[a-zA-Z0-9_]+>

element:  <[a-zA-Z0-9_]+>

%product table.c
```

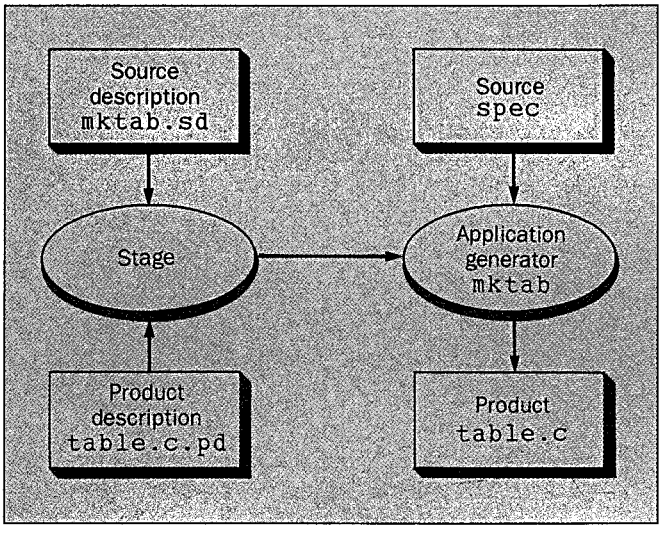

Figure 4. Diagram of Stage.

The `lex` notation is used for describing sequences of characters, called *tokens* (for example, `set_name` and `element`). (`lex` is a popular tool for generating scanners in the UNIX system.) The last line of the specification says that the `mktab` application generator will generate a single product, called `table.c`, whose description can be found in file `table.c.pd`. Panel 5 is a product description using a template approach to define the product of the application generator. Text shown in the constant-width font in this panel represents text that is put directly into the product. Text in the italic font is a metalanguage (all phrases beginning with a percent sign) that describes how to take data from the parse tree and place it in the product.

For example, the phrase, *%d(length(set_def(top)))*, will produce the number of `set_def`s found at the top of the parse tree. The phrase *%(tok(n))* produces the token associated with node *n*. The phrase, *%forall n:set_name %loop*, is used to traverse the parse tree visiting all nodes *n* of type `set_name`.

Stage Architecture. The Stage command reads the source description file and any number of product description files. It creates a subdirectory and generates the code for an application generator in C language. Figure 5 shows sample input and output of the Stage command. One global "header" file is used for parse-tree data-structure definitions and other global information (`global.h`). The main routine coordinates all actions (`main.c`). Stage analyzes the grammar and generates `lex` and `yacc` files. As part of the analysis, Stage will expand abbreviations (such as `set_def+` and `element+ separated-by ","`) and calculate the token follow graph. The token follow graph describes which tokens follow a token according to rules of the grammar. The token follow graph is used to generate a scanner that will only recognize a subset of the possible tokens that depends on the previously recognized token (using `lex` start conditions). With this analysis, we can use generalized tokens, such as `'.*'` and repeated token descriptions (such as in the definitions for `set_name` and `element`).

The parse-tree traversal routines are generated in `trav.c`. The traversal routines are used in the product description file with the *%forall* construct, but may also be used to perform semantic analysis or error checking. For example, to verify that each `set_name` is not a reserved word, one could write a subroutine, `reserved`, and add

Panel 5. Product Description File: `table.c.pd`

```
%template
int  number_of_sets = %d(length(set_def(top)));

char  *name_of_set[] = {
%forall n:set_name %loop
      "%(tok(n))",
%end-loop };

int  size_of_set[] = {
%forall s:set_def
%loop %d(length(element(s))), %end-loop };

%forall s:set_def %loop

char *set_of_%(tok(set_name(s)))[] = {
    %forall e:element in s %loop
      "%(tok(e))",
    %end-loop };
%end-loop

char  **values_of_set[] = {
%forall n:set_name %loop
    set_of_%(tok(n)),
%end-loop };
```

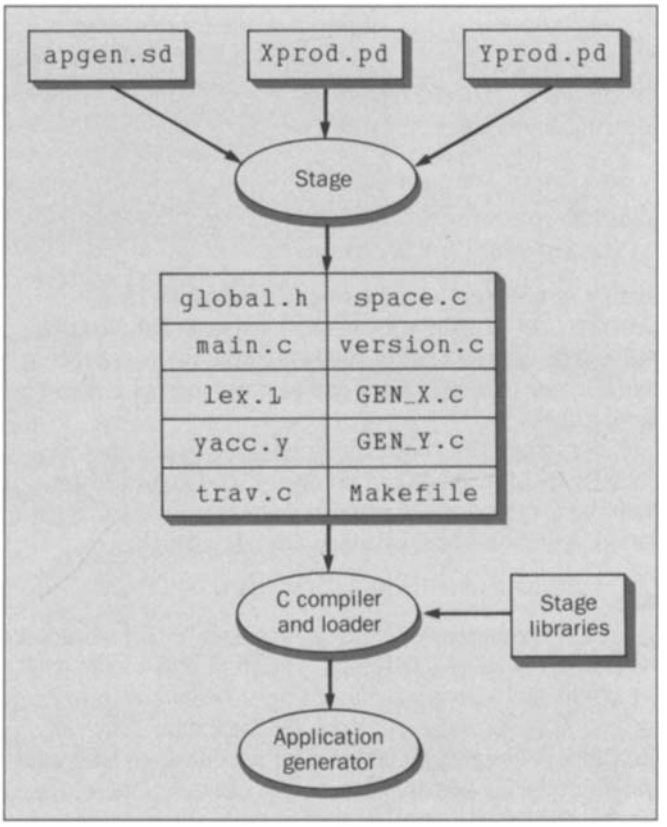

Figure 5. The Stage command.

the code shown in Panel 6.

Stage allows attributes, and by using the parse-tree traversal routines, both *synthesized* and *inherited-attribute* computation rules can be expressed.

Data declarations are generated in `space.c` and `version.c`. Product generation routines for `Xprod` and `Yprod` are generated in `GEN_X.c` and `GEN_Y.c`. Finally a `Makefile` is produced that will build the application generator.

Stage is written in Stage. Thus, there is a file, `stage.sd`, that describes the Stage syntax and product description files for each generated file (such as `main.c.pd`, `Makefile.pd`, and the self-referencing `GEN_Q.c.pd` that describes the translation of product description files to C code).

Stage is designed to produce application generators rapidly. The application generator must be reasonably efficient, but performance is not as critical because application generators are used in limited domains and will not be used as often as other translators such as compilers. The implications of this goal are most apparent in the design of the parse tree.

The parser automatically builds a parse tree based strictly on the grammar. The user merely specifies the language and does not have to think about any other activities that the parser might perform during parsing (such as pop-

Panel 6. Code for subroutine `reserved`

```
FORALL(s,set_name,top,set_name) LOOP
      if (reserved(tok(s)))
         reportwarning(s,"%s is a reserved word.",tok(s));

ENDLOOP
```

ulating symbol tables). The source description thus describes not just the specification language but also the parse-tree data structure. Consequently, the parse tree is much larger than necessary and performance as a whole is not optimal.

In practice, however, application generators produced by Stage usually can produce C code much faster than the C compiler can compile it. In other words, application generators built by Stage are fast enough.

PG2

PG2 generates *only* the back-end code generation parts of application generators. The front and middle parts for syntax and semantic analyses must be built by other means. This decoupling of front and back ends provides flexibility in designing interfaces for specification languages and in producing semantic trees. For example, one can use a form- and menu-oriented input to write the application specifications and do specialized semantic analysis to produce the semantic tree. Obviously, the code generators of PG2 expect their input semantic trees to be in a standard format. Each node is described by giving its level number, the sequence of Boolean-valued attributes of that node that are true, and a list of string-valued attribute name-value pairs for that node.

The semantic tree for our Word Sets example using PG2 is shown in Panel 7. In this Panel, `R` stands for the Boolean attribute "Is Root?" and `L` stands for "Is Leaf?" The absence of a Boolean attribute indicates that that attribute is false for a node. Application generators built using PG2 provide for Boolean attributes A to Z for each node (some letters such as `R` and `L` are reserved; the remaining attributes can be given any semantics in the application).

Panel 7. Input to PG2 for the Word Sets Example Tree in Figure 2

```
0 R
{ name Word_Sets number 3 }
1
{ name colors number 3 }
2 L
{ name red }
2 L
{ name blue }
2 L
{ name green }
1
{ name cities number 2 }
  ...
  ...
```

A PG2-based code generator reads such a semantic tree, builds the corresponding internal tree structure and interprets program abstractions written in a language called KS. A KS-abstraction contains ordinary characters (characters belonging to the target programming language) and directives to the code generator. All directives start with the character ~ (tilde). If the tilde (~) is needed in the target language, ~~ is used. There are three basic kinds of directives. The substitution directive of the form

Panel 8. KS-Abstraction of the Word Sets Example in Panel 3

```
int number_of_sets = ~"number";

char *name_of_set[] = {
~{(<)
        "~""name"",

~}};

int size_of_set[] = {
~{(<) ~"number", ~}};


~{ (<)
char *set_of_~"name"[] = {
~{(<)
        ~"name",

~}
};
~}

char **values_of_set[] = {
~{(<)
        set_of_~"name",

~}};
```

~""name"" asks for replacement of that string with the corresponding string value of that attribute in the current node, or for redirection of input or output. A navigational directive of the form ~{...~} asks the interpreter to interpret recursively the block within the matching braces for each node in the subtree rooted at the current node. A conditional directive of the form ~{(<)...~} asks for interpretation of the inner block only for the first-level children nodes of the current node.

The program abstraction example in Panel 3 written using the KS language is shown in Panel 8. The input tree file and the KS-abstraction file are the only files needed to generate the target program. It is instructive to verify that PG2 interpretation (as described above) of the KS-abstraction in Panel 8 will indeed produce the program in Panel 2. KS language has a rich set of directives for tree traversals (visit all children, visit first-level children, visit the left siblings, visit the ancestor chain of nodes, etc.), conditional navigation using attribute expressions, control structures for branching, iterative and recursive interpretations, and named blocks.

WOODS

The WOODS system is used for building software tools that include:

- Data structures for storing and manipulating information
- Small, flexible, easily-accessible databases for long-term storage of information
- User interfaces and data structure editors for viewing and modifying information
- Translators that transform information to products (including programs).

The basic architecture of a WOODS tool is shown in Figure 6. The user interacts with the WOODS tool through menus, dialogues, and commands. Specifications are stored in data structures that may be edited and stored in external WOODS data files. The specification information is also used to generate products, such as programs and documentation.

Our Word Sets example written using WOODS is given in Panel 9. The first few lines describe a data structure for storing a list of sets; each SET includes a set name and a list of elements. The rest of the file specifies a product generator that will produce the implementation in Panel 2.

The front end of a WOODS tool is much simpler to specify than the source description file of Stage. There are no language design or parsing issues to consider. The end user does not need to learn a language, but simply answers questions in a dialogue. The penalty, of course, is that the information is stored in a rigid data structure and may not be as easy to read as a specification language.

To add or change information, a data structure editor is used. With this editor, the user can move around the data structure to modify various components. The

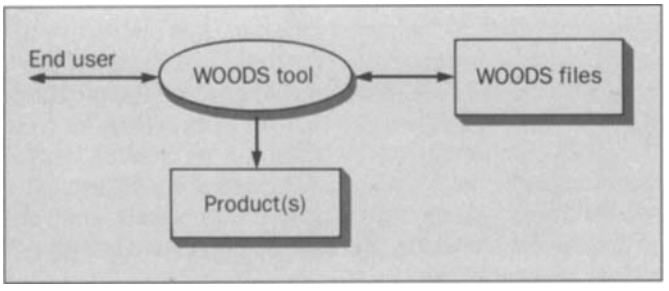

Figure 6. Generic architecture of a WOODS tool.

editing behavior for each object varies. This variation is captured by the concept of properties. Every WOODS data type has a collection of properties that describes the behavior of the object. Properties are organized using an object-oriented inheritance mechanism. They are used to define editing behavior, internal storage, routines for reading and writing data structures to external files, product generation, and many other kinds of operations. A WOODS specification may contain definitions of new types and new properties. Thus, tool builders can customize a WOODS tool to whatever degree they wish.

Panel 9. Word Sets Example in WOODS

```
%type SET is record {
        STRING          set_name;
        STRING_LIST     elements;
        }

%type SETS is SET_LIST

%product table(SETS sets)
%precode pname="table.c";
%begin
int number_of_sets = %#sets.;

char *name_of_set[] = {
%*sets.
        %^set_name.,

%*. };

int size_of_set[] = { %*sets. %#elements., %*. };

%*sets.
char *set_of_%~set_name.[] = %^elements.;
%*.

char **values_of_set[] = {
%*sets.
        set_of_%~set_name.,
%*. };
%end
```

WOODS data structures correspond to attributed parse trees. Like Stage and PG2, WOODS provides ways to traverse the data structures. However, a WOODS back end uses a more advanced metalanguage than Stage product descriptions. Each formatting command begins with a `'%'` and ends with a `'.'`; formatting commands contain operators, identifiers, and escapes to the C language. In Stage, formatting commands specify explicitly where to obtain information, and this usually means a large C expression that explicitly selects many branches of the parse tree. For example, Panel 5 contains the formatting command:

```
%(tok(set_name(s)))
```

The corresponding WOODS format command is:

```
%~set_name.
```

The operator ~ means to figure out what `set_name` means and to print it out on the product. The product generator will look for a name match among the parameters of the procedure, loop variables, and field names of any record it finds. When the appropriate object is found, the properties of the object will give information about how to print it out. Instead of explicit, detailed directions as required in Stage, WOODS allows implicit directions and abbreviations, thus simplifying the product description file. Other WOODS formatting commands are shown in Panel 10. WOODS also lets the user define new formatting commands, a feature lacking in Stage.

Panel 10. Other WOODS Formatting Commands

`%*sets.`	Iterate over the objects in the `sets` list
`%*.`	End of iteration loop
`%#sets.`	Print number of objects in the `sets` list
`%^sets.`	Print `sets` as a C value

Applications and Conclusions

Stage has been used within Bell Laboratories to build a wide variety of application generators for producing software such as user interfaces, database systems, testing programs, control software based on finite state machines, interprocessor communications, hardware device abstractions, and protocols. It has also been used to build programming language tools, such as structured assemblers, and tools that require parsing the C language. Stage has also been used in the hardware development process, including hardware design translators and tools for controlling the design process.

Application generators based on PG2 have been built for a tape data-extraction facility in a tool called TTU (tape-to-UNIX system).[11] PG2 is also fast enough to be used as part of an AT&T internal database query system for large databases. No significant applications have yet been built with WOODS because it is a relatively new tool.

With the introduction of Stage, some of the barriers to using application generators have been lowered. Stage is a well documented, stable tool that has a three-day training course available. This lowers the initial barrier to understanding, building, and applying application generators. As a result, many application generators have been built that would not have been otherwise.

The technology to build application generators is not new. However, the techniques are not well known and are seldom applied. As with any new technology, there is difficulty in transferring these techniques to many projects. People are reluctant to consider new techniques because it is risky; it means learning something new; and it requires initiative, creativity, and energy. The field needs a *champion*, one who promotes the idea of creating and using application generators.

Acknowledgments

The authors and Dave Belanger (co-inventor of PG2) appreciate the many people who were early users and champions of our tools, particularly Rick Greer, Terry Hansen, Dave McDonald, Dave McKay, and Griff Smith.

References

1. P. Bassett, "Design Principles for Software Manufacturing Tools," *Proceedings of ACM Annual Conference*, Association for Computing Machinery, October 8-10, 1984, pp. 85-93.
2. J. M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering*, Vol. 10, No. 5, September 1984, pp. 564-573.

3. A. Milton Jenkins, "Surveying the Software Generator Market," *Datamation*, September 1, 1985, pp. 105-120.
4. F. Van Hoeve and R. Engmann, "An Object-oriented Approach to Application Generation," *Software—Practice and Experience*, Vol. 17, No. 9, September 1987, pp. 623-645.
5. J. Craig Cleaveland, "Building Application Generators," *IEEE Software*, July 1988, pp. 25-33.
6. T. F. Payton et al., "SSAGS: A Syntax and Semantics Analysis and Generation System," *Proceedings of COMPSAC'82*, Computer Software and Applications Conference, 1982, pp. 424-433.
7. J. Bentley, "Programming Pearls: Little Languages," *Communications of the ACM*, Association of Computing Machinery, Vol. 29, No. 8, August 1986, pp. 711-716.
8. M. Ganapati, C. Fischer, and J. Hennesey, "Retargetable Code Generators for Compilers," *ACM Computing Surveys*, Association for Computing Machinery, Vol. 14, No. 2, December 1982, pp. 573-592.
9. C. M. R. Kintala, "Attributed Grammars for Query Language Translations," *Proceedings, 2nd ACM Symposium on Principles of Database Systems*, Association for Computing Machinery, March 1983, pp. 137-148.
10. S. E. Keller et al., "Tree Transformation Techniques and Experiences," *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction, SIGPLAN Notices*, Vol. 19, No. 6, June 1984, pp. 190-201.
11. D. G. Belanger and C. M. R. Kintala, "Data Extraction Tools," *AT&T Technical Journal*, Vol. 64, No. 9, November 1985, pp. 2025-2035.

Biographies (continued)

computer science from Pennsylvania State University. He is also an Adjunct Professor at Stevens Institute of Technology.

(Manuscript received April 26, 1988)