SOFTWARE—PRACTICE AND EXPERIENCE. VOL. 22(3). 223–244 (March 1992)

The Design and Implementation of a Grammar-based Data Generator

PETER M. MAURER
ENG 118, Department of Computer Science and Engineering, University of South Florida, Tampa. FL 33620, U.S.A.

SUMMARY

DGL is a context-free grammar-based language for generating test data. Although many of the features of DGL are implemented in a straightforward way, the implementation of several of the most important features is neither trivial nor obvious. Before one can understand the implementation of these features, however, it is necessary to understand the overall structure of the compiler and its output, which was designed to be flexible enough to incorporate new features easily. Variables and chains are two of the most important features of DCL, and also two of the trickiest features to implement. The run-time dictionaries, which are built into the C code generated by the compiler, are implemented as pure code rather than as table-look-up routines. The compiler itself is reasonably straightforward, except for the expansion of character sets and compile-time macros. These two features can cause the 'multi-dimensional' expansion of a string, the implementation of which must be carefully designed.

KEY WORDS Testing Compilers Context-free grammars

INTRODUCTION

DGL (data generation language) is a language for automatically generating test data.[1–3] DGL is based on the concept of probabilistic context-free grammars,[4] and allows data to be generated either systematically or randomly, or some combination of the two. The purpose of this paper is to describe the implementation of the DGL compiler and the generated code. As in any complex program, some parts of the DGL system are straightforward, whereas other parts are quite complex, and not at all obvious. Rather than trying to explain every detail of the DGL system, this paper will concentrate on those aspects of the generated code and the compiler that represented especially difficult design problems. Armed with this knowledge, the rest of the DGL implementation should be obvious to the reader.

DGL is designed to be a tool for facilitating the generation of test data for random testing of software. Oddly enough, the most extensive application of this tool has been in the area of design-verification of VLSI circuits.[5] Although VLSI design verification appears to have very little in common with software testing, such verification is usually done using a software simulation of the circuit rather than the circuit itself. Because of this the differences between software testing and VLSI design verification have all but disappeared.

The problem of generating tests for VLSI circuits is particularly challenging

0038–0644/92/030223–22$11.00
© 1992 by John Wiley & Sons, Ltd.
Received 28 March 1991
Revised 23 September 1991

because such circuits often have complex inter-chip protocols that must be obeyed if the circuit is to function correctly. The behaviour of such circuits is often unspecified if the correct protocol is not observed. This implies that purely random input will be of little value in verifying the correctness of such circuits. What is needed is a test-generation method that allows certain portions of the input to remain fixed while other portions are generated at random. For the circuit described in Reference 5, it was necessary to generate a series of ‘bus transactions’, each one of which had to obey a specific protocol, but contained some random elements. Since these bus transactions could be arranged in several different ways, it was necessary to be able to perform a second level of random selection to construct sequences of bus transactions.

Another problem with the design verification of VLSI circuits is that, at least in some sense, 100 per cent coverage of the circuit is required. At the very least, all microinstruction must be exercised. Such coverage is difficult, if not impossible, to achieve using purely random testing. On the other hand, it is well known that 100 per cent path coverage of a problem does not guarantee that the program is correct, and this is no less true for VLSI circuits than for software. Therefore, one of the initial design objectives of DGL was to allow one to begin with a set of tests known to exercise all paths in a design, and randomize them in a controlled way.

The concept of context-free grammars lends itself quite well to these problems. To randomize a test, one simply replaces the fields to be randomized with non-terminal symbols. Productions are then used to describe the format of the randomly generated data. The data-generator can then read the test description and replace the non-terminal symbols with randomly generated data. Suppose, for example, that the test description contains the non-terminal symbol `B`, and that a production of the form `B–>xyz` has been specified. A specific test can be generated by replacing the symbol `B` with the right-hand side of the production, `xyz`. If more than one alternative exists in the production, as in `B–>abc|xyz`, then a random number generator is used to make a choice between the two alternatives. Since the right-hand side of a production can contain non-terminals, multi-level selection is automatically available. To complete the picture, the test itself can be specified as an alternative for the start symbol of the grammar.

The original version of DGL contained only two types of productions: ordinary productions and weighted productions. Ordinary productions are those such as the production `B–>xyz`. When a choice must be made between several alternatives in an ordinary production, all choices are equally likely. A weighted production is identical to an ordinary production, with the exception that a numeric weight is assigned to each alternative. When alternatives are chosen, those with higher weights will be chosen more often than those with lower weights. DGL has been heavily used for VLSI design verification ever since its inception, and has been instrumental in debugging several complex VLSI designs at the University of South Florida. Not only have most users found it to be easy to use, once a general introduction has been given, but they have also been quite eager to incorporate DGL-based random testing into their methodologies. As yet there has been very little experience with using DGL to test end-user software, but there is no reason to suppose that DGL would be any less useful for this purpose than for testing VLSI simulators.

Although today’s version of DGL is considerably more complex than the original, most of the new features were added specifically at the request of a user. For

example, several users expressed the need to 'generate the same thing twice', to write and then read a randomly generated memory address, for example. To solve this problem, variables were added to DGL. Variables can store a randomly generated string, and allow it to be inserted into the output at several different places. Several other users expressed the need to generate data systematically. Many different features were provided to solve this problem, not all of which survive in the current version. The most powerful of these is chains, which permit the systematic generation of highly complex data.

Several other researchers have investigated the problem of generating test data using extended context-free grammars. One such system is described in Reference 2. Many of these systems are oriented toward the testing of compilers, probably because of the ready availability of the test grammar, but there is no reason why such systems cannot be used for other types of software. DGL provides many more features than previous grammar-based systems, and integrates its features in a more versatile way. For example, the system described in Reference 2 can generate data either randomly or systematically, but two different data generators are used. This makes the integration of random and systematic data generation difficult or impossible. DGL allows systematic and random data generation to be mixed arbitrarily.

There are other automatic test-generation systems that are based on finite state machines and regular grammars.[6] These test-generation systems have an advantage over DGL in that the expected results of a test can be generated automatically along with the test data. However, the program under test must be described as a finite state machine, which limits the usefulness of such systems. Although DGL places no restrictions on the structure of the program under test, some automatic method of examining test results is usually needed for DGL to be used most effectively. Because DGL grammars can be integrated with separately written subroutines, it is possible to add an 'expected results' field to each test, even for complex tests. However, the responsibility for computing expected results lies with the user.

LANGUAGE DESIGN

The fundamental component of a DGL test specification is the production which has the following form:

```
name: choice_1 ,choice_2, ..., choice_n;
```

Each of the choices is a string of characters, possibly constructed out of several components, which can contain both terminal symbols and non-terminals. A non-terminal is a string of the form % {name}, and a terminal symbol is anything else. There are several syntactic short-cuts that can be used to specify choices. First, excessively long strings can be broken at any point and specified on separate lines. This is a special case of the rule that two strings with no intervening comma are concatenated into a single string. The DGL compiler ignores spaces, tabs and newlines, except as separators between syntactic items. As a warning to future language designers, this rule has caused no end of headaches, since a missing comma between two choices is not considered to be a syntactic error, but produces something quite undesired by the programmer. It would have been better to have required that a specific concatenation operator be specified.

Most strings may be specified with or without quotes. The only exception is when certain special characters appear in the string, in which case the string *must* be quoted. Quoting removes the meaning from most special characters, except the % of the non-terminal. To remove the meaning from this character it is necessary to specify two consecutive % characters. This has also proved to be a point of great confusion to DGL users, but one that is not so easily rectified. The difficulty lies in the fact that non-terminals are identified interpretively at run time rather than at compile time. Furthermore, a DGL grammar may deliberately build a non-terminal out of individual characters, store it in a variable, and then reference the variable, which will cause the non-terminal to be recognized as such and processed accordingly. For this reason it was necessary to have a simple rule for removing the meaning from % that could be applied at run time without resorting to the complex syntactic rules used by a compiler.

As a consequence of these rules, the following four specifications mean exactly the same thing:

```
"abc%{def}ghi"
"abc""%{def}""ghi"
abc  %{def}ghi
abc  %{  de  " f} " g  hi
```

It is also possible to include a set of characters in a string specification as in [0–9] or [abc]. When such a specification appears unquoted in a string, it causes the rest of the string to be copied several times, once for each element of the set, and successive elements of the set to be inserted into the string in place of the character-set specification. For example, the following two specifications are identical:

```
abc[def]
abcd,abce,abcf
```

The dash is used to specify a range of characters. The specification [0–9] is identical to [0123456789]. Ranges are interpreted in the underlying character set of the machine on which DGL is running, so different machines can give different results.

Macros are the string equivalent of a character set. One declares a macro by inserting the keyword macro into an ordinary production. A macro reference has the form !name. A macro reference is treated similarly to a character set, except in this case the choices from the referenced macro are used to replace the macro reference in the successively created strings. The following two specifications are identical:

```
abc: macro a1 ,b2, c3
def: xyz !abc;

def xyza1 ,xyzb2,xyxc3;
```

Macros must be defined before they are used. This rule was added to permit DGL to be compiled with a single-pass compiler, and to eliminate the need for circularity checks in macro references. (Macro references can be resolved as they are encountered, so as far as the compiler is concerned, nested macro calls do not exist.)

Current DGL implementations allow any ordinary production to be used as a macro whether or not it was declared as such. The macro declaration is used only to suppress generation of selection code for the macro. This decision may be changed in future DGL implementations, because the right-hand sides of all ordinary productions must be saved in case the production is used as a macro later.

The primary problem with macros and character sets is that a single choice can have several macro references or character sets, requiring the sort of multi-dimensional expansion illustrated by the following two identical productions:

```
abc: [xy][xy];
abc: xx,xy,yx,yy;
```

In addition to ordinary productions, DGL provides several other types of productions which alter the rules by which choices are made from the alternatives on the right-hand side of the production. For ordinary productions, a random choice is made from the right-hand side. Other productions allow some choices to be selected with higher probability than others, as in the following example:

```
abc: 1:a,2:b,1:c;
```

This production will cause b to be selected twice as often as a, or c, on the average. The alternatives of the following production will be chosen sequentially in the order specified, rather than at random:

```
abc: sequence a,b,c,d,e;
```

One particular useful type of production is the variable, which is declared as follows:

```
var1 : variable " This is the initial value ";
```

A variable acts as if it were an ordinary production with a single alternative, except that the content of the alternative may be changed by dynamically using non-terminals of the form %{abc.var1}. Non-terminals of this form consist of two production names separated by a period. The second production must be a variable, but there are no restrictions on the type of the first. This non-terminal produces no output, but causes the value selected from abc to be assigned to var1. When the non-terminal % {var1} is specified, the current contents of var1 are inserted into the output at the point. If var1 contains non-terminals, they will be expanded appropriately. The following illustrates how to insert non-terminals into a variable:

```
main:xyx%{abc,var1}qed%{var1};
abc:%%{z};
z: zzzzz ;
var1: variable;
```

The non-terminal %{main} is the start symbol for this grammar. Interpreting %{main} will produce the output xyxqedzzzzz. The main problem with variables is that

the output of any production can be redirected into a variable, and that assignments can be nested.

DGL provides many productions similar to the sequence production illustrated above which allow data to be generated systematically. By far the most complex production for systematic data generation is the chain production, because it requires that selections from several different productions be co-ordinated, and there are very few limitations on how these productions can be used. Any finite set of strings can be generated systematically by writing a grammar for it, and inserting the chain keyword into each of the productions. The only restriction is that one can use only ordinary productions, otherwise the structure of the grammar is quite unrestricted. For example, suppose it is necessary to generate all strings that consist of three alphabetic characters followed by four digits. (This is a large set containing 175,760, 000 members.) The following grammar can be used to generate this set:

```
main: chain %3{a}%4{n};
a: chain [a–z];
n: chain [0–9];
```

In this example, the production %3{a} is equivalent to %{a}%{a}%{a}. This example requires selections to be co-ordinated between the three productions. It is also possible to mix chain productions with other types of productions in a reasonably unrestricted way. Suppose that in the above example, it was desired to generate all alphabetic combinations systematically, but choose the four-digit number at random. This can be accomplished by removing the chain keyword from the n production.

For most types of productions, the actions that must be taken when expanding a non-terminal are obvious, and in many cases quite trivial. However, deciding when to perform these actions and determining where to put the output is more difficult. It is important to realize that the DGL compiler converts a DGL grammar into a C program. When compiled, this C program constitutes the data generator that was described by the grammar. Certain parts of the generated code are required to perform dictionary look-ups for symbols encountered during run-time processing. Because linear look-ups of these symbols began to severely degrade the performance of DGL data generators when many productions were specified, 'pure code' dictionaries were designed to enhance the performance of these operations.

GENERATED CODE STRUCTURE

The right-hand side of an ordinary production consists of a number of strings, each of which may contain a number of terminals and non-terminals. A typical string might be Cde%{xyz} 123%{iy}%d, where %{xyz}, %{iy} and %d represent non-terminals, and Cde and 123 represent terminals. Since the right-hand sides of ordinary productions are known at compile time, it would certainly be possible to interpret them there rather than at run time. However, when the DGL compiler was first designed, it was anticipated that 'production-like' strings could be created dynamically either by external subroutines or by new types of productions that did not at that time exist. Because a run-time interpreter would be required anyway for such strings, it was decided, for simplicity, to interpret all strings at run-time rather than compile time. Interpreting a string consists of making a left-to-right scan of the

chosen string, outputting or storing all terminal symbols and replacing all non-terminals with choices from the referenced productions. The algorithm illustrated in Figure 1 is a simplified version of that used by all DGL data generators.

This algorthim omits the details of processing truncated non-terminals such as %{abc} or %{.xyz}, and of non-terminals with repetition counts such as %5{abc} and %3-7{xyz}. Furthermore, error processing for undefined productions and variables is also omitted. The run-time interpreter contains more than one hundred lines of code, and is therefore too complicated to be replicated for each production. Therefore it is implemented as a subroutine which is called by the production handling code.

The next problem was determining the relationship between the run-time interpreter and production-handling code. One alternative would have been to integrate the production handlers into the interpreter itself. However, when the interpreter was first designed, it was anticipated that it might be necessary to add many new types of productions. (This later turned out to be the case.) It would be necessary to integrate the code for these new productions into the existing compiler without affecting the code generation for existing productions. As a mechanism for isolating the code for different types of productions, the decision was made to incorporate all code for a particular production into one or more subroutines that were specific to that particular production. Every production would have a selection routine which would be called to make a selection from its right-hand side. Variables would have assignment routines in addition to their selection routines. The interpreter would have no knowledge of production types, and would treat every production identically. This scheme allowed all the peculiarities specific to a particular type of production to be isolated from all other code. Furthermore, because C subroutines may be used recursively, implementing each production as a subroutine allowed the structure of the code generator to mirror the natural recursive structure

```
intepret(string)
begin
    /* Process string from left to right*/
    while There are characters left to process do
        if There is no unprocessed non-terminal
            output all unprocessed characters;
        else
            output all unprocessed characters up to the next non-terminal;
            NTNAME:= the name of the next non-terminal;
            if NTNAME contains a period then
                Move the variable name from NTNAME to VARNAME;
            else
                VARNAME:= the empty string;
            endif
            if VARNAME ≠ the empty string then
                Assign a selection from the non terminal named in NTNAME
                    to the variable named in VARNAME;
            else
                Make a selection from the non terminal named in NTNAME;
            endif
        endif
    endwhile
end
```

Figure 1. The interpreter algorithm

of a context-free grammar. Figure 2 gives an example of a selection routine for an ordinary production. Variable assignment routines are discussed in greater detail in the section entitled ‘Variables’.

The problem of distinguishing between declared and undeclared productions and variable names was solved by providing two look-up routines, similar to the hash-table look-up routines used in most compilers. If a production has been declared, the look-up routine will return the address of the production’s selection routine, otherwise it will return NULL. These look-up routines were implemented as two separate dynamically generated subroutines so that the run-time interpreter could be implemented as a precompiled library routine in a future version of DGL.

The final problem was that of redirecting output. Potentially, the output of any production can be redirected into a variable. Although the interpreter subroutine performs all output for most productions, there are some types of productions (such as counters) that perform output in their selection routines. Since output can be performed in many different places, and since redirecting output is somewhat messy, it was desirable to isolate the redirection of output into a single subroutine rather than distribute it throughout the generated code. For this reason a basic output routine was developed which replaced the C putc routine. This routine, which is ultimately responsible for performing all output, can be switched back and forth between the standard output and an internal storage area by altering the contents of a global variable. Several higher-level output routines were designed around this low-level routine, in much the same way that higher-level C output routines were developed around putc.[7] The overall structure of a DGL data generator is illustrated in Figure 3. As this Figure illustrates, data is generated by making one or more calls to the production handler of the grammar’s start symbol.

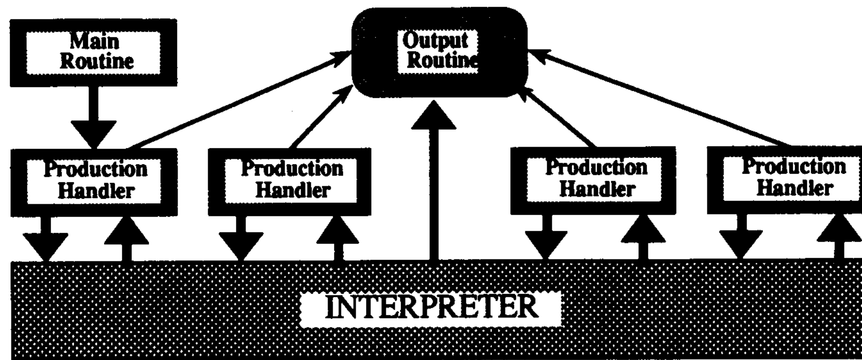
In addition to the selection routine, each production has a collection of C variables which contain the value(s) of the right-hand side of the production as well as the current state, if applicable. To illustrate this further, Figure 4 illustrates the code generated for a sequence production which has an internal state. To simplify the implementation of state variables, all internal states were designed to be represented either as single integers or as pointers to data-structures. When the state is represented as a pointer to a data structure, it is normally necessary to make a copy of the data structure when saving the state of the production.

```
abc:   qed,abc%{def}hij,13245;

Declare abc_value to be an array containing the
three strings "qed", "abc%(def)hij", and "13245".

subroutine abc_select;
begin
       i:= A Random Number from 1 to 3;
       Interpret(abc.values[i]);
end
```

Figure 2. A .simple selection routine



Figur e3. The structure of a code generator

```
abc:  sequence  qed,abc%{def}hij,13245;

Declare abc.value to be an array containing the
three strings "qed", "abc%(def )hij", and "13245".

Declare abc_state and initialize it to 1.

subroutine abc_select;
begin
        i:= abc_state;
        abc_state:= abc_state + 1;
        if abc_state >3 then
                abc_state:= 1;
        endif
        Interpret(abc_values[i]);
end
```

Figure 4. A simple selection routine

CHAINS

From the beginning it was clear that DGL needed a mechanism for exhaustive generation of certain sets of data, although it was not clear exactly what this mechanism should be. It *was* clear that for anything other than trivial sets of data, some co-ordination of selections from various different productions would be necessary. One early approach was to replace the conventional interpreter with one that co-ordinated selections from several different productions. However, most practical applications of exhaustive test generation required a mixture of exhaustive techniques and random selection. For example, it might be necessary to generate HLL compiler tests of the form X: =Y+Z, where X, Y and Z range over all data types, with the values of Y and Z generated randomly. Because of this mix, it was usually necessary to customize the new interpreter manually for a specific application.

Another early approach was to provide a co-ordinating production in which the right-hand sides of certain subordinate productions were also specified. The selection routine for this production could then co-ordinate the selections from each of the subordinate productions. This approach was more general purpose than customized

interpreters, but severely limited the types of grammatical structures that could be used to describe exhaustively generated tests.

In a sense, both of these early approaches violated the original spirit of the DGL implementation, namely that all code specific to a particular production should be isolated inside that production's selection routine. Although it seems difficult to reconcile this spirit with the need to co-ordinate selections from several different productions, the chain construct was the final result of doing just that. Surprisingly, the chain construct also turned out to be both very powerful and remarkably easy to use.

Syntactically, the chain production is simply an ordinary production containing the chain keyword. The implementation of chains is based on the data structure

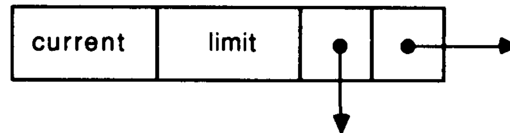


Figure 5. A data structure element for implementing chains

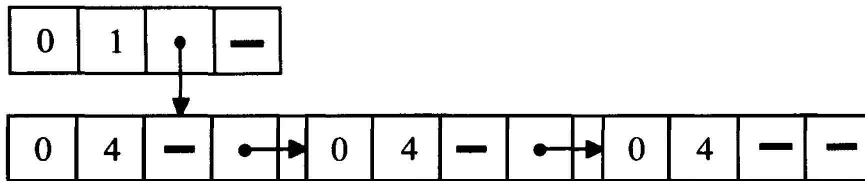
illustrated in Figure 5. This data structure, which is created dynamically, represents the current state of one production in a co-ordinated set of chain productions. The current field contains the number of the next alternative to be chosen, and the limit field contains the total number of alternatives for the production. The data structure contains two pointers called right and down, respectively. The down pointer points to the data structure for the leftmost non-terminal of the current alternative. The right pointer is used to create a linked list of data-structures for non-terminals that appear in the same alternative.

To clarify the use of the chain data-structure, consider the grammar illustrated in Figure 6(a) which generates all three-letter strings containing the letters a, b, c and d. The first string generated by this grammar is aaa, which is produced by the data structure illustrated in Figure 6(b). The string dac is produced by the data structure illustrated in Figure 6(c). Note that production alternatives are numbered from left to right starting with zero.

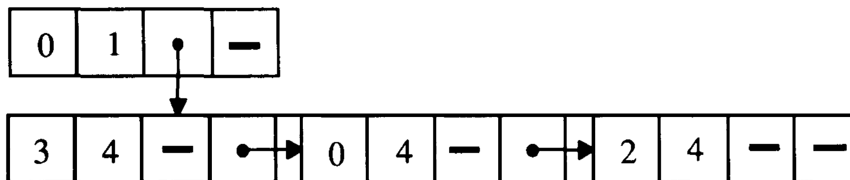
The chain data structure is used to create a standard binary-tree representation of the derivation tree of a string. This tree is considered to be the current state of the production that appears at the *root* of the tree, and only chain productions are represented in the tree. Creation of the derivation tree is done by the selection routines using two global variables named parent and left. When a chain selection routine is called, parent will point to the parent of the current production, and left will point to the production to the left of the current production on the same level of the tree. When parent is NULL, the selection routine will either begin building a new derivation tree, or obtain an old derivation tree from the state variable of the current production. If left is NULL, the data structure for the current production is pointed to by the down pointer of parent, otherwise it is pointed to by the right pointer of left. When a new derivation tree is being created, the field pointing to the structure for the current production will be NULL. When this occurs, a new data-structure is allocated and chained into either left or parent as appropriate. When a new data structure is created, the current field will be set to zero, and the right and

```
exam: chain %{let}%{let}%{let)\n;
        let  chain  a,b,c,d;
```

a. A sample chain grammar.



b. The Data Structure for generating "aaa".



c. The Data Structure for generating "dac".

Figure 6. The use of the chain data structure

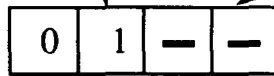
down fields will be initialized to NULL. Figure 7 illustrates the construction of the tree illustrated in Figure 6(b).

Once the selection routine has obtained (or created) its own data structure, it uses the current field to select an alternative, and passes the corresponding string to the interpreter. Before calling the interpreter, it saves the current value of parent, assigns NULL to left, and makes parent point to its own data structure. When the interpreter returns, the selection routine restores the value of parent, makes left point to its own data structure, and then returns to its caller (normally the interpreter). The chain selection routine is illustrated in Figure 8.

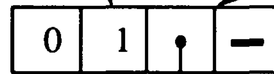
If the restored value of parent is null, then this is the root of the derivation tree, and it is necessary to advance the derivation tree to generate the next string. This is done by calling the special advance routine pictured in Figure 9. The strategy used by the advance routine, is to first attempt to advance the production immediately to the right, if that fails then try to advance the production immediately below, and if that fails then try to advance the current production. An attempt to advance a NULL pointer always fails. When an attempt to advance either the right or down pointer fails, the corresponding data structure is destroyed and the pointer is assigned the NULL value. The current structure can be advanced only when both left and down and NULL. The destruction of the down pointer is necessary on failure because a new alternative will be chosen for the current production, and the structures of the new and old alternatives may not be identical. Destruction of the right pointer is not strictly necessary, but is done for the sake of uniformity in the algorithm. When

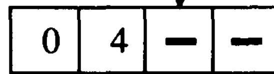
Beginning of First Call to "exam_select"
Parent: NULL Left: NULL exam_state: NULL

Beginning of First Call to "let_select"
Parent: ● Left:NULL exam_state; ●

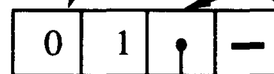


Beginning of Second Call to "let_select"
Parent: ● Left: ● exam_state: ●





Beginning of Third Call to "let_select"
Parent: ● Left: ● exam_state: ●



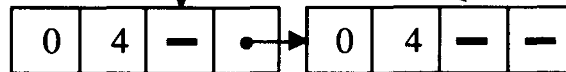


Figure 7. The creation of a chain derivation tree

the advance fails at the head of the chain, all possibilities have been exhausted, and several user selectable actions may be taken.

Because all processing of the chain data structure (except advancing) is incorporated into the selection routines, chain productions may be freely mixed with other types of productions. When chain productions are used, it will usually be the case that the number of different strings generated by the grammar is finite. However, with a properly constructed grammar, it is possible to enumerate the first *n* strings of an infinite set. For example, the following grammar enumerates the regular set {a,b}*:

```
string: chain %{letter}, %{string} %{letter};
letter: chain a,b;
```

In the second alternative of the string production, the order of the non-terminals is important, since the rightmost non-terminal will be varied the Fastest. If the order

```
abc_select()
begin
    local_parent:chain_structure;
    current_node:chain_structure;

    if parent = NULL
    then
            current_node := abc_state;
            if current_node = NULL then
                    current_node:= new_node(0,production. alternative_count)
                    abc_state := current_node;
            endif
    else-if left = NULL then
            current_node:= parent ↑ .down;
            if current_node = NULL then
                    current_node:= new_node(0,production. alternative_count);
                    parent ↑ .down:= current_node
            endif
    else
            current_node = left ↑ .right;
            if current_node = NULL then
                    current_node:= new_node(0,production. alternative_count);
                    left ↑ .right:= current_node
            endif
    endif,
    current_alternative:= abc_values[current_node ↑ .current];
    local_parent:= parent;
    left:= NULL;
    parent:= current_node;
    interpret(current_alternative);
    left:= current_node;
    parent:= local-_parent;
    if parent= NULL then
            left:= NULL
            if NULL advance(current_node) then
                    production.state:= NULL
            endif
    endif
end
```

Figure 8. The selection routine algorithm for chains

```
advance(current_nodechain_structure):boolean;
begin
        if current_node = NULL then return FALSE endif;
        if advance(current_node ↑ .right) then return true
        else current_node ↑ .right:= NULL endif;
        if advance(current_node.down) then return true
        else current_node.down := NULL endif;
        current_node.current:= current_node.current + 1;
        if current_node.current ≥ current_node.limit
        then
                current_node.current:= 0;
                return false
        endif;
        return true
end
```

Figure 9. The advance algorithm for chains

of these non-terminals were reversed, the grammar would generate only the following strings: a, b, aa, ab, aaa, aab, aaaa, aaab, and so forth.

Chains do have drawbacks, one of which is that a chain cannot be referenced inside another chain. For example, suppose it is necessary to generate tests systematically using a set of chain productions, and it is necessary to number each test with a four-digit hexadecimal number using the production hexnum as illustrated below:

```
test: chain %{hexnum} ...;

hexnum: chain %4{hexdigit};
hexdigit: chain [0-9], [a-f];
```

If this grammar is used without alteration, first every test will be generated and numbered zero, then every test will be generated and numbered one, and so forth. This is clearly not what is intended. The head production was introduced to solve this problem. The head production is a special type of chain production that always acts as if it were the root of the derivation tree. Because of this, an independent derivation tree is created for all head productions. Before head productions were introduced, it as possible to solve this problem by using variables. First, the production hexnum would be assigned to a variable, and then the value of the variable would be used to create the test as illustrated below:

```
test2: %{hexnum, hexvar}%{test};
test: chain%{hexvar} . . . ;
```

A second drawback of chains is that each successive string is generated by repeating a sequences of choices. The derivation tree will force the choices for chain productions to be identical, but if part of the derivation tree was created based on a random selection from a non-chain production, there is no guarantee that an identical choice will be made for successive strings. For example, consider the following grammar:

```
hexnum: chain %{hexnum2};
hexnum2: %{hexdigit},%{hexnum2}%{hexdigit};
hexdigit: chain [0-9], [a-f];
```

The first time that hexnum is interpreted, production hexnum2 will make a random number of choices from hexdigit. The second time that hexnum is interpreted a different number of choices from hexnum2 will probably be made. However, the intent is (apparently) to make a random number of choices from hexnum2, and then exhaust all of the strings that can be made from that number of choices. Fortunately, the practical applications of this sort of grammar are rare, but it is still possible (although not *easy)* to program around the problem.

VARIABLES

When the interpreter encounters a non-terminal of the form %{xyz.abc}, it will call the assignment routine for the variable abc rather than the selection routine for xyz. The assignment routine for a variable has one parameter, which is the name of the

```
abc_assign(NonTerminalName)
begin
        Save the current output status;
        Redirect output into a temporary storage area;
        Call the selection routine for NonTerminalName;
        Delete the current contents of abc, if any;
        Create a string from the data in temporary storage;
        Assign the newly created string to abc_state;
        Restore the current output status;
end
```

Figure 10. A variable assignment routine

production from which the new value of the variable is to be chosen. The structure of a variable assignment routine is illustrated in Figure 10.

In addition to its assignment routine, every variable also has a selection routine, which is illustrated in Figure 11.

There are two problems that can occur in connection with variable assignments. First, the value assigned to a variable can be quite large. In general, the size of a selection from a non-terminal is both unlimited and unpredictable. Secondly, variable assignments can be nested, as in the following example:

```
abc: qed%{s1.v1 }the;
s1 : grx%{s2.v2}eno;
```

To solve these problems, the temporary storage area for accumulating variable values is implemented as a collection of dynamic data structures, as illustrated in Figure 12.

Figure 12(a) illustrates a single storage element which is used to hold data for variable values. Each element holds a fixed number of characters. (In the current implementation, this number is 100.) Initially, a single block is allocated, but if the number of characters exceeds the limit, an additional block will be allocated and linked to the first using the Next Horizontal field. Additional blocks can be added, creating a singly linked list as illustrated in Figure 12(b). A global pointer, current, points to the head of the list. The field Last Block points to the most recently allocated block, which is the block to which the next character will be added. This creates a horizontal list, such as those illustrated in Figure 12(b).

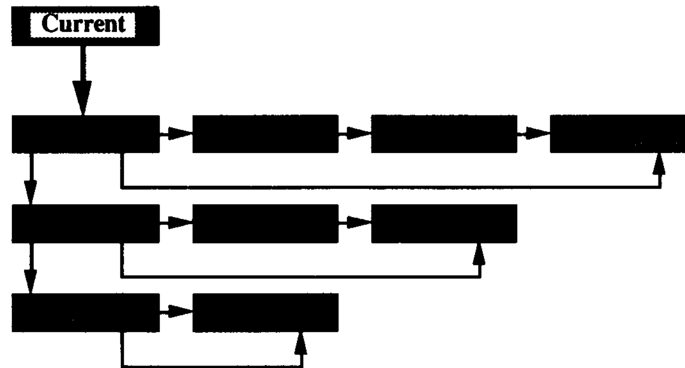
The vertical structure illustrated in Figure 12(b) is created by nested variable assignments. When a new variable assignment is begun, a new block will be allocated and pushed onto the stack which is implemented using the global pointer current and the Next Vertical field of the first block in each horizontal list. When a variable

```
Declare abc_state to be a pointer
    to a dynamically allocated string;

abc_select()
begin
        interpret(abc_state);
end
```

Figure 11. A variable selection routine

a. A single storage element.



b. A dynamic storage structure.

Figure 12. The dynamic storage area for variable values

assignment terminates, the top horizontal list in the stack is popped off, the number of characters in the list is counted, a string is allocated to hold the data, all data is transferred from the horizontal list into the string, and the data structures of the horizontal list are deallocated. This method of handling variable assignments allows nested assignments to work properly even, in some cases, when the nested assignments are to the same variable. Because the old value of the variable is not destroyed until the last possible moment, operations such as concatenating characters to the end of a variable will work properly.

RUN-TIME DICTIONARIES

The interpreter uses two run-time dictionaries, one for selection routines and one for variable assignment routines. The look-up of a particular entry is done by a series of binary searches which are completely unrolled. The first binary search is done on the length of the input string. The subsequent searches (if any) are done on successive characters of the input string. Conceptually, the dictionary is organized as a hierarchical collection of sublists. First, the collection of all entries is broken into sets of strings such that all strings in a particular set have the same number of characters. Each set of strings is further broken down into subsets such that the strings in each subset all begin with the same letter. Each subset is broken down further using the second character of each string, and so forth. This break-down continues until, at the lowest level, each subset contains a single string. Successive binary searches are used to select sublists successively until only a single string remains. Then the right-hand portion of the string that was not used for selection is

compared to the corresponding right-hand portion of the selected entry. Like the binary searches, this final comparison is also completely unrolled.

Because the binary searches and final comparisons are completely unrolled, no data structures are required for storing the dictionary entries. The look-up routine consists of a collection of nested if statements. To clarify the structure of the look-up routine, suppose that a grammar contains production names with lengths 3, 5, 9 and 12. The binary search on string length is illustrated in Figure 13.

To illustrate the subsequent searches, assume that the strings of length 5 are ggccc, gmddd, mgeee, and mmfff. The code to search these strings is illustrated in Figure 14. In Figure 14, it is assumed that the input string is stored in an array named | whose lowest index is zero.

Although Figures 13 and 14 illustrate the unrolled loops as nested if statements, it was necessary to implement them using labels and go-to statements, because our version of the C compiler was incapable of handling such deeply nested control statements.

The unrolled search routines have recently replaced linear search routines for the selection and assignment routines. The use of these routines has made the larger code generators run noticeably faster. In the near future we plan to implement similar routines for state variables and for restoring the state of productions from an external file.

COMPILER STRUCTURE

For the most part, the DGL compiler is a straightforward parser implemented using the YACC preprocessor. [8] As soon as a production is parsed, code is generated for it. Two temporary files are used to hold the generated code, one for declarations of global variables, and one for selection and assignment routines. If the production is an ordinary production or a macro, it right-hand side is saved for referencing by subsequent productions. Otherwise, all compiler data structures for the production are destroyed once code has been generated.

Most productions contain a list of strings as part of their right-hand side, the general syntax of which is given in Figure 15. As this Figure illustrates, a string can be created by concatenating a sequence of simple strings, macro references and

```
if input_length >5 then
     if input_length >9 then
          if input_length ≠ 12 then return NULL;
          else search strings of length 12; endif
     else
          if input_length ≠ 9 then return NULL;
          else search strings of length 12 endif
     endif
else
     if input_length >3 then
          if input_length ≠ 5 then return NULL;
          else search strings of length 5; endif
     else
          if input_length ≠ 3 then return NULL;
          else search strings of length 3; endif
     endif
endif
```

Figure 13. An unrolled binary search on string length

```
if I[0] > "g" then
    if I[0] ≠ "m" then return NULL;
    else
        if I[1] > "g" then
            if I[ 1 ] ≠ "m" then return NULL;
            else
                if I[2]="f" and I[3]="f" and I[4]="f" then return mmfff_select;
                else return NULL; endif
            endif
        else
            if I[1] ≠ "g" then return NULL;
            else
                if I[2]="e" and I[3]="e" and I[4]="e" then return mgeee_select;
                else return NULL; endif
            endif
        endif
    endif
else
    if I[0] ≠ "g" then return NULL;
    else
        if I[1] > "g"then
            if I1] ≠ "m" then return NULL;
            else
                if I[2]="d" and I[3]="d" and I[4]="d" then return gmddd_select;
                else return NULL, endif
            endif
        else
            if I[1] ≠ "g" then return NULL;
            else
                if I[2]="C: and I[3]="C" and I[4]="C" then return ggccc_select;
                else return NULL; endif
            endif
        endif
    endif
endif
```

Figure 14. Searching a set of four strings

character sets. Before code can be generated for a production, it is necessary to convert strings containing character sets and macro references into sets of simple strings. The first step is to convert the string specifications into binary trees whose vertices represent concatenation operations and whose leaves represent strings, macro references and character sets. Figure 16 shows the structure of such a tree. In this Figure S represents a string, R represents a macro reference and C represents a character set. A similar binary tree is used to represent a character set, but in this case the vertices of the tree represent the dashes that appeared in the original character-set specification.

The processing for character sets and macro references is quite similar. The macro reference causes a list of strings to be fetched from the compile-time dictionary, and the character set causes a list of one-character strings to be generated from the specification. The string-creation algorithm transforms trees such as that pictured in Figure 15 into lists of strings by traversing the binary tree and gathering data into a fixed-size accumulation area. Macro references and character sets are handled via back-tracking. To facilitate this, the string-creation routine is implemented as a recursive procedure with two arguments: a tree-vertex and the current offset into

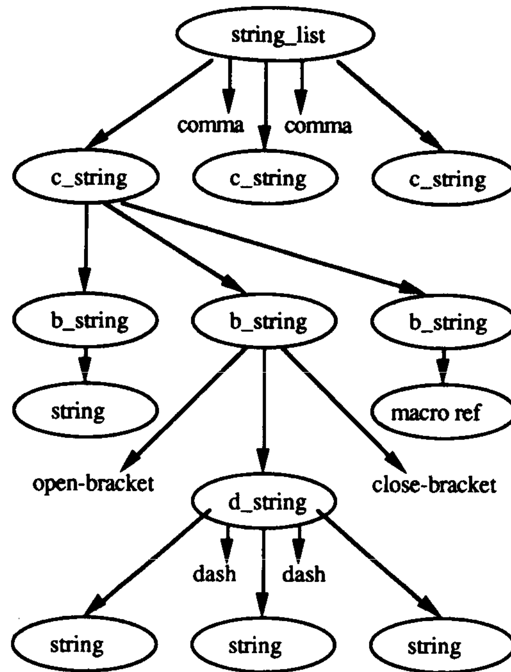


Figure 15. The grammatical structure of the RHS of a production

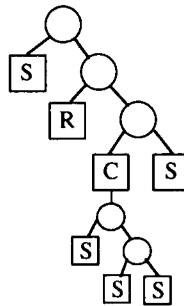


Figure 16. The structure of a parsed string

the accumulation area. The initial call to this routine is done using the root of the tree structure and an offset of zero.

When the string-creation routine processes a non-leaf vertex, it processes the left child (which is always a leaf) and calls itself recursively on the right child. If the left child is a string, it is copied into the accumulation area starting at the current offset. The offset for the recursive call is equal to the current offset plus the length of the left-hand side. When a character set or macro is processed, each string in the list is copied successively into the accumulation area starting at the current offset. (The current offset does not change from one string to the next.) A recursive call to the right child is done for each string.

The processing for leaves is identical to that for non-leaf vertices, except that no recursive call is made. Instead, the routine adds the contents of the accumulation area to the current list of strings.

The main limitation of this technique is that the size of the accumulation area is fixed at 5000 characters, although in practice it is unlikely that this limit would be exceeded.

PERFORMANCE EVALUATION

Several experiments were performed to measure the performance of DGL; however, because performance is highly dependent on the structure of the grammar, these experiments can give only a rough idea of what the performance will be on a specific grammar. All experiments were run on a SUN4 IPC with 12 megabytes of memory.

The first experiments was designed to compare the performance of chains with that of random test generation. The following grammar was used, both with and without the chain keyword. Each version of the grammar was used to generate 100, 000 tests. In this case, the chain version of the grammar actually outperformed the random selection version. The random selection version consumed 30 CPU seconds, whereas the chain version consumed only 28 CPU seconds:

```
main: chain % 10x\n;
x: chain 0,1,2,3;
```

The grammar used for the first experiments generates a two-level parse tree. To determine the performance of chains with deeper parse trees, the following grammar was used:

```
main: chain %\n;
s: chain %x, % s%x;
x: chain 0,1,2,3;
```

This grammar will generate strings similar to those generated by the first grammar, but of varying lengths. The test generator constructed from this grammar required almost 90 CPU seconds to generate 100,000 tests. The storage requirements for the parse-trees were modest in both grammars. The first grammar generates a static a parse-tree with 11 elements. Since each element is 16 bytes long, the total storage consumed was 176 bytes. For the second grammar the parse tree grows as more test are generated. Generating the 100,000th test required a parse-tree with 17 elements, for a total of 272 bytes.

Two additional grammars were created to measure the performance of the DGL compiler, and to analyse the performance and size of the run-time look-up routines. The first grammar consisted of 256 productions as illustrated below. Most test grammars contain fewer than 256 productions, and use production names that are easier to distinguish than these. The DGL compiler generated 8197 lines of C code for this grammar, and consumed approximately 1 second of CPU time doing so. The breakdown of the generated code is 2382 lines for data structures, 1792 lines for production processors, 423 lines for utility routines, and 3600 lines for the run-time look-up routines.

```
aaaaaaaa: a,b, c;
aaaaaaab: a,b, c;
aaaaaaba: a,b, c;
aaaaaabb: a,b, c;
...
bbbbbbbb: a,b,c;
```

Because the run-time look-up routines are completely unrolled, they contain only comparisons and return statements. On a conventional architecture, these comparisons consist of two instructions, a compare followed by a conditional branch. For the grammar illustrated above, only one comparison is needed to check the length of the input string, but in general $\lceil \log_2 n \rceil + 1$ comparisons are required, where n is the number of different length production names specified in the grammar.

In addition to the length comparison, 765 character comparisons were generated. Identification (or rejection) of any input string requires a maximum of 16 of these comparisons to be executed. For a linear search, an average 128 string comparisons will be performed to identify a string. A successful comparison requires eight character comparisons, and on the average, two comparisons will be required for each unsuccessful comparison (the mathematical analysis which yields to this value is quite complicated). Therefore, a linear table look-up algorithm will require an average of 264 character comparisons for each item. After compilation with the SUN4 C optimizer, the look-up routine consumed 13 Kbytes of storage. The complete generated code consumed 30 Kbytes, 2 Kbytes for data structures and 28 Kbytes for instructions.

CONCLUSION

Although DGL has become extremely complex, in a sense, it is still in its infancy. This paper has touched on only a few of the many different types of DGL productions, and there are certainly many new types of productions that could be added. For example, it might be useful to have productions that were capable of performing arithmetic. Careful study will be required to determine which new features will be most useful.

DGL has already proved to be useful in generating tests for VLSI designs, [5] and may prove to be equally useful in generating tests for software. Nevertheless, the experience with DGL as a language is quite limited compared to most other programming languages. Would it be useful to make DGL into a full-flown general-purpose programming language? The answer to this question is not yet clear. On the other hand, it is certain that DGL will continue to evolve as experience with the language and its implementation grows.

ACKNOWLEDGEMENT

This work was supported by the University of South Florida Center for Microelectronics Research (CMR).

REFERENCES

1. P. M. Maurer, 'Generating test data with enhanced context free grammars'. *IEEE Software,* **7,** (4), 50–56 (1990).
2. A. G. Duncan and J. S. Hutchinson, 'Using attributed grammars to test designs and implementations', *Proc. 5th Int. Conf. on Software Eng.,* 1981, pp. 170–178.

3. D. C. Ince, 'The automatic generation of test data', *Comput. J.,* **30,** 63–69 (1987).
4. T. L. Booth and R. A. Thompson, 'Applying probability measures to abstract languages", *IEEE Trans. Comput. ,* **C-22,** 442–450 (1973).
5. P. M. Maurer. 'The design verification of the WE 32100 math accelerator unit'. *IEEE D & T,* **5,** (3), 11–21 (1988).
6. J. A. Bauer and A. B. Finger, 'Test plan generation using formal grammars', *Proc. 4th Int. Conf. on Software Eng.,* 1979, pp. 425–432.
7. B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style,* McGraw-Hill, New York, 1974.
8. S. C. Johnson, 'YACC: yet another compiler compiler', *Computing Science Technical Report Number 32,* Bell Laboratories. Murray Hill, NJ, 07974, 1975.