DGL

Generating Test Data with Enhanced Context-Free Grammars

Peter M. Maurer

Generating Test Data with Enhanced Context Free Grammars

Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

ABSTRACT

Enhanced context free grammars are an effective means of generating test data, even for simple programs. Grammars can be used to create intelligent random samples of tests as well as to create exhaustive tests. Action routines and variables may be used to compute expected results and perform other actions that are difficult or impossible to perform with ordinary context free grammars. Grammars may be used to create subroutine test data that contains pointers and other binary data. Test grammars not only provide a method for improving software quality, but should prove to be the foundation of much future research.

Generating Test Data with Enhanced Context Free Grammars*

Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

1. Introduction

When asked to name their ten most useful programming tools, most programmers would probably not place context free grammars at the top of the list. In fact for most of us, the term brings to mind a collection of obscure proofs from some barely remembered theory of computation class. Even if context free grammars were a magic balm that could somehow make the most bug-ridden program run correctly, many programmers would be too intimidated by the underlying theory to use them. Nevertheless, my experience using context free grammars to generate tests for VLSI circuit simulators has convinced me that they are remarkably effective tools that can be used by virtually anyone to debug virtually any program.

Although it is not obvious, VLSI testing and software testing have much in common. VLSI circuits have become so complex that more time is spent testing their functionality than is spent testing their electronics, in most cases a great deal more. Because functionality testing is usually done on a software simulation rather than on the circuit itself, the distinction between software testing and VLSI testing has all but disappeared.

I discovered the usefulness of context free grammars while I was looking for a way to improve the quality of functional tests for VLSI circuits. This was not a trivial task, because our methodology was already very good. We monitored our simulators to guarantee that every line of code was executed. We included code to monitor assertions and report violations. We analyzed control paths to guarantee that each one had been executed. We identified boundary conditions and tested them exhaustively. And we conducted many code inspections. In spite of this, the bugs would just not go away, and irritatingly enough, our users seemed to be able to find them quite readily. (I have had many similar experiences with many other kinds of software.)

---

* This work was supported by the University of South Florida Center for Microelectronics Design and Test

Because we had exhausted all of the systematic methods known to me, I decided to experiment with random samples of tests taken from the problem domain. Now, random testing is not a new idea (For an overview of automatic test generation see [1] and its many references. For an approach similar to the one described here, see [2].) Random tests have been used to test both software and hardware with similar results. Random tests can be used to find the easy bugs (or hardware faults) but finding the difficult bugs requires more systematically generated tests. What I wanted was a method for generating tests that was substantially more clever than a simple random number generator. I wanted a system that would take a template describing the format of a test and generate a collection of random tests according to the template. I wanted to begin with a test that was designed to test for a difficult bug (for example) and generate many different tests to detect the same bug and hopefully many others. After spending a lot of time thinking about the underlying principles of such a system it occurred to me that what I had in mind was an interpreter for context free grammars. Development of the tools proceeded rapidly once I had this concept firmly in mind.

My first test generator was dedicated to testing a single VLSI chip (or more correctly a software simulation of the chip), and proved to be surprisingly effective[3]. Since that time I have developed several "data-generator generators" that translate context free grammars into test generators. The most useful of these is "dgl" (for data generation language) developed at the University of South Florida. Dgl and its underlying language (which for lack of imagination I also call "dgl") are vastly different from that first data generator, but the underlying principle is the same. The data generator contains a grammar that describes the tests, and when it is executed it generates one or more tests according to the grammar. Although originally intended as a method for generating random tests, dgl has evolved into a tool that can also be of assistance in generating some kinds of systematic tests. At present dgl is still evolving. Every new application of the tool uncovers new ways the tool could be used "if only it had this new feature." Nevertheless, the underlying principle of using grammars to generate tests remains firm, and this is the subject of the rest of this article.

2. Before You Begin to Test

Although the methods described in this paper will enable you to generate a thousand tests as easily as one, you must develop a careful plan both for generating the tests and for using them. You must start with a methodology that you know is sound. (For example you should have some method of guaranteeing that every line of code is executed at least

once.) The methods described here are designed to enhance the effectiveness of a sound methodology, not to replace it.

Your most immediate problem will be determining whether a test has passed or failed. If it takes you an hour to analyze the results of one test, it will take you a thousand hours to analyze the results of a thousand tests. You must establish some method of quickly deciding whether a test has passed. This can be done in several ways. You may wish to construct tests that are easy to diagnose. If you can see at a glance whether a test has passed, you can run and analyze a thousand tests with very little effort. Of course in such a case, a simple analysis program could analyze the results of a test in microseconds, enabling you to run not thousands but millions of tests.

My personal preference is to generate self-diagnosing tests, that is tests that contain an expected result, and use an analysis program to determine whether the test has passed. To do this, you must either have tests whose outcome is easy to predict, or you must have more than one version of the program you are testing. In the first case, it is easy to adapt the test-generation grammar to predict the result of the test. The second case is more complicated. In some applications it is the usual practice to have more than one version of a program available. For example in VLSI testing there is usually more than one software simulator available, and these are usually quite different from one another. In life support systems, and other systems designed for high reliability, it is common to have two or more different versions of a program. In other cases, there may be a fast prototype or a previous release available. At times it might be worthwhile to write a simplified version of the program for the sole purpose of predicting test outcomes. In any case, you should test both programs independently and verify the results manually to guarantee that you're not just comparing one bug against another. I have not found the problem of generating expected results to be especially restricting. I believe it is a good policy to provide all tests with expected results and check the results automatically, regardless of where one obtains them. Even if it is necessary to verify the results of each test manually, context free grammars can be used to simplify the construction of the required tests.

One aspect of testing for which I have found grammars to be ideal is performance measurement. Since one typically does not care about the outcome of such tests, one can ignore the problem of generating expected results. Furthermore, because one is concerned with the average case rather than testing all special cases, the problem of detecting the hard bugs is not of primary importance.

3. Your First Tests

The best way to create a test grammar is to start with a set of existing tests and turn it into a grammar. Suppose you have written a calculator program to do simple arithmetic, for which you have created the tests pictured in Figure 1.

```
2+1
2+-1
3-2
3*7
3*8
4*7
7/3
4/2
```

Figure 1. Tests for a Calculator Program.

You have two add statements because your parser handles negative numbers differently from positive numbers. You have several divides and multiplies because your arithmetic routine handles powers of two by doing shifts instead of multiplies and divides. Figure 2 shows how your first grammar might look.

```
test:   %{number}+%{number},
        %{number}+-%{number},
        %{number}-%{number},
        %{number}*%{number},
        %{number}*8,
        4*%{number},
        %{number}/%{number},
        %{number}/2;
number: 0,1,2,3,4,5,6,7,8,9;
```

Figure 2. A First Test Grammar.

(In this and all other examples I follow the dgl convention of specifying productions in the form "<name>: <choice-1>, <choice-2>, ... , <choice-n>;" and nonterminals in the form %{name}.) Notice that this grammar preserves the properties that made the original test good. There are examples of multiplying and dividing by powers of two and there are examples containing negative numbers. We could make this grammar even better by changing the "number" production to generate positive and negative multi-digit numbers and by adding a production to generate powers of two. If we have several sets of tests, we can combine the grammars in many different ways. The most obvious is to simply add more alternatives to the "test" production of the above grammar. We could also use a

"parent" production of the form "test: testset_1, testset_2, ..., testset_n;" and make "testset_i" the start symbol of the ith set of tests.

At this point it is appropriate to say something about how the above grammar can be used to generate a test. The start symbol is "test" so first select one of the alternatives from the "test" production. Let's assume we choose the alternative "%{number}+%{number}." We will scan this alternative from left to right and replace the nonterminals with data. When we find the first nonterminal "%{number}" we will choose an alternative from the production "number" and use that to replace the nonterminal. Eventually we will end up with a string such as "7+4." If we want to generate a second test, we begin again by making another choice from the production "test." Since we may choose the alternative "%{number}+%{number}" many times, we must do our replacements on a copy of the alternative rather than on the alternative itself.

The key question is how to make a choice from a set of alternatives. In the above example, dgl will choose alternatives at random with an equal probability of choosing any particular alternative. For many applications, this may not be the best way of choosing alternatives. At the very least you may want certain choices to be made more often than others, for example in the above you may want to concentrate on multiplication and division and limit the number of tests for addition and subtraction. In more complicated examples you might want to avoid choosing the same alternative twice, or you might want alternatives to be chosen in a particular order. To handle these problems, dgl provides selection rules that specify the way alternatives are to be chosen and allows weights to be assigned to alternatives to force some to be chosen more often than others. Indeed, much of the research that has gone into dgl has been concerned with identifying and implementing useful selection rules. An exhaustive discussion of these rules would be, to say the least exhausting, so I will limit my discussion to those that I believe are the most useful and natural.

Probably the most natural selection rule is specifying the probability of selecting a particular alternative. There is already a probability associated with each alternative, why not let the user change it? One place where I find this particularly useful is in generating tests for VLSI arithmetic circuits. Some arithmetic algorithms are sensitive to the number of ones and zeros in their arguments, so using arguments with only a few ones or only a few zeros tends to find bugs. The following example generates binary numbers with only a few zeros and lots of ones.

```
bnumber: %32{bit};
bit: 31:1, 1:0;
```

On the average, only one out of 32 selections from "bit" will be a zero. (I've snuck a bit of dgl shorthand into this example. The nonterminal %32{bit} means to make 32 consecutive selections from "bit.") The idea of weighting the alternatives of a production is not new. See [4] for a complete discussion of probabilistic context free grammars. The most obvious use of weights is to guide the test generator to favor those tests that you feel will detect the most bugs. In performance measurement weights can be used to construct a random sample of tests with a given mix of test types. An intriguing opportunity for future research might be to keep track of the number of bugs detected when a particular alternative is chosen and automatically adjust the weights to favor those alternatives that detect the most bugs.

To summarize this section, I believe that the best place to start constructing a test grammar is from tests that are known to be good, and work in a bottom-up fashion. It is also possible to work in a top-down fashion by constructing a grammar that describes every possible test. Unless this method is used carefully you may end up with a grammar that has a very low probability of generating tests for the "hard" bugs. On the other hand, I have found that a sprinkling of purely random tests is sometimes useful for finding categories of tests that are important but somehow got overlooked. For example, in the calculator program will "divide by shifting" work if the dividend is negative?

4. Action Routines

Another natural extension to context free grammars is to add action routines to certain alternatives. There are many ways to do this, but in dgl I have chosen to define an action routine as an arbitrary subroutine written in the C language. Action routines provide a handy method for computing the expected results of a test, provided the tests are not too complicated, and they can be used to implement esoteric functions not available in the dgl language. To see how action routines can be used to generate expected results, let us return to the calculator example. Suppose we want to generate test data and expected results in the following format.

<value><operator><value> = <expected result>

For simplicity I will show only the addition alternative "%{number}+%{number}." Figure 3 shows a grammar that will do the job.

```
test: %{number_x}+%{number_y}=%{add_result};
number_x: %{number.x}%{x};
number_y: %{number.y}%{y};
x: variable;
y: variable;
number:  0,1,2,3,4,5,6,7,8,9;
add_result: action (
        output(x_value+y_value);
);
```

Figure 3. An Example of an Action Routine.

The contents of the action routine is implementation dependent, but most reasonable implementations would provide similar features. The two productions "number_x" and "number_y" are used to make the two most recent choices from the "number" production accessible to the action routine. The nonterminal %{number.x} makes a choice from "number" and assigns the choice to the variable "x." Because the nonterminal %{number.x} produces no value, the nonterminal %{x} is needed to insert the value of the choice into the test. The action routine adds the two most recently selected values and uses the "output" function to insert the result into the test.

I leave the more esoteric uses of these routines to your imagination, but let me say a bit more about variables. There are many kinds of tests that require you to use the same value many places. When constructing a grammar for such tests, it may be necessary to generate such a value at random. Variables allow you to do this, as the following example illustrates.

```
double:  %{number.x}%{x}+%{x}
x: variable;
number:  0,1,2,3,4,5,6,7,8,9;
```

In this example, a number is chosen at random, assigned to the variable "x," and then the contents of "x" is inserted into the test twice. You might use such a test to verify that an optimization of the calculator program for doubling a number works correctly.

The main point of variables is that they enable you to do things that cannot be done without them. In a theoretical sense, context free grammars with variables are more powerful than ordinary context free grammars (they are, in fact, *universal* in a theoretical sense). It is my belief that *any* grammar-based system to generate test data must rely on some extension of context free grammars that enhances their theoretical power. Many such extensions have been developed, but of all those with which I am familiar, I find variables to be the easiest to implement and to use. (There is always room for debate on such issues.)

Variables may contain both data and nonterminals. This is useful when the *size* of the test depends on data contained in the test. Such examples are rare, but they do occur in practice. It is sometimes easier to use action routines to generate this kind of data, but the The grammar of Figure 4 shows how variables may be used to do the job.

```
test: %{number.x}"The following number contains " %{x} "digits: "%{val};
number:  0,1,2,3,4,5,6,7,8,9;
x: variable;
val: %{gen.y}%{y};
gen: "%%" %{x} "{number}";
y: variable;
```

Figure 4. Storing a Nonterminal in a Variable.

In this example, the production "val" first assigns the value "%n{number} to the variable "y," and then inserts the value of "y" into the test. Since "y" contains a nonterminal, its contents are replaced by n selections from the production "number." (The double "%%" is dgl syntax that keeps the replacement from being done until *after* the value is placed in the variable.)

5. Generating Systematic Tests

Although I first began using grammars to generate random tests, it rapidly became clear that they could also be used to simplify the generation of systematic tests. In my work it is often necessary to test many different special cases sometimes alone, and sometimes in combination with other special cases. Furthermore, I have come to rely almost exclusively on self-diagnosing tests and automatic methods for reporting test failures. Under these circumstances, writing a large number of tests is far more time consuming than running them or analyzing the results. One alternative would have been to design a second data generator that would generate *all* tests described by a grammar, instead of making choices at random. (This approach has been used by Duncan and Hutchison[2].) However, many of my special cases were of the form "a 32-bit number beginning with 011" and that sort of thing. I didn't want to choose specific values for fields that did not require them, and I didn't want to generate a huge number of tests just to test one special case. I eventually discovered a method for generating all possibilities for some productions while allowing other choices to be made at random.

The implementation is quite complicated and beyond the scope of this paper, but the *use* of this feature is quite simple, as the example of Figure 5 shows. This example

generates tests for the "multiply-by-shifting" optimization of our calculator program. All specified powers of two are tested exhaustively with randomly selected numbers.

```
test:       chain     %{number} * %{power_of_two},
                      %{power_of_two} * %{number},
                      %{power_of_two} * %{power_of_two};
number:               0,1,2,3,4,5,6,7,8,9;
power_of_two:  chain  1,2,4,8,16,32,64,128,256;
```

Figure 5. Generating Exhaustive Tests.

The only difference between this and a purely random example is the "chain" keyword on the productions named "test" and "power_of_two." Since the production named "number" does not have this keyword, its choices are made randomly rather than systematically.

I believe that a similar feature would be useful in any grammar based system for generating tests, but one word of warning. Such features generate huge numbers of tests. Make sure you have some efficient method for determining whether they pass or fail.

I have found other systematic selection methods to be useful in generating both random and exhaustive tests. Two of these are generation of sequence numbers, and selecting alternatives from a production sequentially rather than randomly. These both work pretty much as you would expect them to, so I won't burden you with examples.

6. Generating Tests for Subroutines

Although it has always been my ambition to test complicated subroutines independently, I have usually been put off by the difficulty of creating data for them. The most complicated subroutines always seem to process complex structures that contain pointers and binary data. Even generating simple input of this nature usually requires a complex program. Recently I was attempting to debug a complicated macro processor (for generating VLSI structures) and I decided to try to apply my work in test grammars to the problem of debugging one particular routine that was giving me a lot of trouble. This routine was designed to evaluate arithmetic and logical expressions that had been parsed into a tree-like data structure whose nodes are pictured in Figure 6.

Figure 6. An Expression Node.

The field "type" contains an integer that specifies the operation to be performed, while the fields "left" and "right" point to the operands. To simplify things, assume that a "type" of 1 means addition, 2 means an integer operand, and 3 means a variable operand. For addition "left" and "right" point to the data structure elements defining the operands. For integer operands, "left" contains the value of the operand and "right" is null (zero). For variable operands "left" contains a pointer to the variable name, which must be a single letter a-g and "right" is null (zero). The grammar of Figure 7 may be used to generate these structures.

```
expr: %{add_type}%{expr_pointer}%{expr_pointer},
        %{number}%{value}%{null},
        %{variable}%{var_name}%{null};
  add_type: binary 1;
  number: binary 2;
  variable: binary 3;
  expr_pointer: pointer %{expr};
  value: binary 0,1,2,3,4,5,6,7,8,9;
  var_name: pointer a,b,c,d,e,f,g;
  null: binary 0;
```

Figure 7. Generating Complex Data Structures.

Notice that this grammar is not too different from the grammars used to generate other kinds of tests. The only difference is that some productions return binary values and pointers instead of character strings. I incorporated data generator produced by this grammar into a simple driver routine (about 10 lines) that called the data generator and passed the data structures to my subroutine. To be truthful, this is not the grammar I used to debug my subroutine. I actually used a grammar that generated several specific tests rather than one that generated tests randomly. Using this approach I found I was able to "get into the guts" of the subroutine far more quickly than I could testing the program as a whole. Furthermore, since preparing the test grammar took only a few minutes (for each set of tests) I was able to debug my subroutine in a very short time.

Although I believe that this work is on the right track, I am not yet satisfied with the results. At present it is easy to generate C-language structures, but I'm not sure about other languages. Furthermore, the style seems a little too dependent on the underlying implementation of the data structure. More work is needed in this area. One possible approach would be to couple a grammar-based test system with an interactive symbolic debugger that understands the peculiarities of the various languages, and is capable of transforming a more generic specification into structures suitable to a particular language.

7. A Word of Warning

The example of the preceding section contains something that I have been careful to avoid in all other examples: a recursive grammar. There is a hidden danger in using recursive grammars to generate data. This danger is of a theoretical nature, not an implementation problem (dgl likes recursive grammars just fine). To illustrate, consider the recursive grammar of Figure 8.

```
exp:    %{exp}+%{exp},
        %{exp}*%{exp},
        (%{exp}),
        %{variable};
variable: a;
```

Figure 8. A Recursive Grammar.

The four alternatives of the production "exp" are chosen with equal probability. When a nonterminal of the form %{exp} is replaced, the replacement has a 50% chance of having two copies of %{exp}, both of which must eventually be replaced. It has a 25% chance of containing one copy and a 25% chance of containing no copies of %{exp}. Now, think about the string that the data generator is currently expanding, and count only the occurrences of %{exp}. The string has a 50% chance of getting bigger, a 25% chance of staying the same size and only a 25% chance of getting smaller. The data-generator will not stop until the size is zero. In short, your test might be infinitely long. To get this grammar to work, you must weight the fourth alternative so that the probability of getting smaller is greater than the probability of getting larger. If you use this rule of thumb with all of your recursive grammars you should stay out of trouble.

For a complete theoretical analysis of this phenomenon, see reference [4].

8. Conclusion

I have found the enhanced context free grammars described in this paper to be very effective tools for generating test data of many different kinds. I have used these grammars to debug many different programs with great success. Due to the large volume of tests that can be generated from a test grammar, it is usually necessary to devise some automatic method for predicting the outcome of a test and diagnosing the results once it has been run. For some tests, action routines can be used to compute the expected outcome, but a separate program is usually needed to compare the actual result with the predicted result and report discrepancies.

I have used grammars to debug many different programs, and I am continually surprised by their effectiveness. For example, I was recently explaining the principles of test grammars to a student using a simulator for a binary division circuit as an example. I showed him how to construct the grammar, how to generate expected results, and how to automatically check for test failures. Before I ran the tests I said, "Of course I'm just using this circuit as an example. It's much too simple to benefit from such a powerful testing technique." In the next instant over 70% of the tests failed. The circuit did not work for unsigned numbers with a high order digit of one.

Nevertheless, the tests are no better than the grammar used to generate them. I believe that there is much room for additional research in this area. One avenue might be to investigate methods for generating test grammars automatically from program specifications, or from the code itself. Regardless of future developments, the ability to create a large number of tests with a minimum of effort makes test grammars an effective tool for improving software quality.

REFERENCES

1. D. C. Ince, "The Automatic Generation of Test Data," The Computer Journal, Vol. 30, No. 1, 1987, pp. 63-69.

2. A. G. Duncan, J. S. Hutchison, "Using Attributed Grammars to Test Designs and Implementations," Proceedings of the Fifth International Conference on Software Engineering, 1981, pp. 170-178.

3. P. M. Maurer, "The Design Verification of the WE 32100 Math Accelerator Unit," IEEE Design and Test of Computers, June 1988, pp. 11-21.

4. T. L. Booth, R. A. Thompson, "Applying Probability Measures to Abstract Languages," IEEE Transactions on Computers, Vol. C-22, No. 4, May 1973, pp. 442-450.