AUTOMATED SYNTAX TESTING USING JSYNTEST™

Syntax testing, also called grammar-based testing, is a powerful black box, data-driven testing technique for testing applications where the input data can be described formally. Originally developed to help test compilers and language processors, syntax testing can also be used in applications that are more mundane.

Here are some possible domains:

1. **GUI applications.** GUI applications typically involve user interaction via dialog boxes. These dialogs often have data fields (e.g. date, phone number, social security number) that have a precise syntax.
2. **XML/HTML files.** All XML/HTML files have a precise structure with well-defined tags. Such a structure is amenable to formal specification.
3. **Command-driven software.** These are among the common applications that benefit from syntax testing. Shell scripts and non-GUI applications that take command line arguments are examples.
4. **Scripting languages.** Some commercial applications are bundled with support for a scripting language such as **Perl**, **Python**, **VBScript™**, and so on. For example, our tools **JStyle™** and **JEvolve™** support scripting in **VBScript™** and **JMScript™**. Script support enables users of the application to extend its functionality in arbitrary ways. These scripting languages have a precise, formally expressible syntax.
5. **Database query languages.** Database query languages such as SQL can be described by a formal grammar.
6. **Compilers.** Testing compilers is a formidable task due to the size of the input space. Since the compiled language has a precise grammar, it should be feasible to use the syntax specification to generate test cases, just as it is used for syntax validation.

The Approach

When we follow syntax testing to generate test cases for a problem, we typically represent the input space using a formal syntax, similar to BNF. In cases where the target language is implicit, additional effort is required to mine the language before coming up with a grammar. Defining the grammar satisfactorily might require a couple of iterations. Since data generation is driven by the grammar, we should strive for completeness and consistency. From the grammar, we systematically generate test data. In many cases, we will have to generate "dirty data", data that does not satisfy the grammar. This is required if we need to assure that the component under test handles invalid data correctly.

The approach to data generation using syntax testing is the reverse of what a language parser does with the grammar. Take the case of *Yacc*. *Yacc* allows us to express the grammar of a language in a specific syntax and generates a parser for that language. The generated parser is typically a C program (other language versions might be available) that when compiled and run, is capable of recognizing any string that conforms to the specified grammar. To handle context-sensitive situations, Yacc allows us annotate the grammar with action code, which get appropriately embedded in the generated code. In syntax testing, however, an automated tool such as **JSynTest** generates a synthesizer, instead of a recognizer, from the grammar.

Automated Tools

Although syntax testing can be performed manually, it is impractical when attempted on problems with a large input space characterized by complex relationships. In such cases, the availability of a tool that can automatically generate the test data from a formal specification will be a major advantage. Of course, the hard task of defining the grammar has to be performed manually, but the tool can help by supporting at least minimal debugging.

JSynTest

At **Man Machine Systems**, we have built a Java-based syntax-testing environment. The following were our design goals:

- The representation formalism must be sufficiently rich to capture a wide range of input data.
- To facilitate ease of use, the tool should sport a GUI.
- To assist testers in defining the input space as accurately as possible, at least minimal support for debugging the grammar must be available.
- It should be possible to build grammars incrementally.
- The environment must encourage and support reuse of grammars.
- It should be possible to embed the data generation logic within an application.

Since we were keen on building a framework that was also portable, Java appeared to be a natural choice.

The JSynTest Approach

Using **JSynTest** is a three-step process:

Step 1. Describe the input data space.

Step 2. Generate a Java program from the input description.

Step 3. Compile and run the generated Java program.

Synthesizing a Java program from the input specification, as opposed to directly emitting the data set, was a conscious decision so as to support embedding the data generation logic in an application. There is a runtime library to support the logic contained in the generated code. As in any program development effort, the above steps are repeated until the generated data set meets our expectation.

Describing the Data Space

Since the ability to specify the data space accurately is crucial to automated generation, we spent substantial effort to arrive at an acceptable formalism. Our formalism is similar to BNF, with enhancement to support user-defined actions.

In **JSynTest**, the data space is described in one or more grammar files. Here are the salient features of our formalism:

- Grammar can be composed from other grammars.
- Grammar can be inherited for reuse.
- A grammar can optionally be qualified as *abstract* or *final*.
- Non-terminal (NT) nodes can be overridden in a derived grammar.
- NT nodes can be qualified as *final* to prevent overriding.
- Actions can be associated with each NT node if needed.
- Grammar object can be *reflected* upon.
- User-defined actions will be preserved across code regeneration.

Grammar File

A target language is described in one or more grammar files. A grammar file is a text file written to conform to a special syntax understood by **JSynTest**. The following are the top-level constituents of a grammar file:

- A qualified grammar name,
- structure section,
- actions section (optional) and
- bindings section (optional).

Every grammar is named. In addition to serving as an aid in denoting the target language it describes, it can be used for building other grammars. It should be in a file that has the same root name as the grammar name itself.

The structure section defines the structural relationship between various non-terminals and terminals of the target language. The derivation rules are enclosed within a pair of braces after the structure keyword.

A rule consists of a left-hand side (LHS) and a right-hand side (RHS). The symbol on the LHS is referred to as the non-terminal symbol (NT). A terminal symbol is one that does not have an associated RHS, and is considered a leaf node. The RHS can be made up of non-terminals and terminals combined using "&" ("and") and "|" ("or") logical operators, qualified by repetition markers. The following are some rules:

```
A: B;
B: C & "D" & C | E;
C: "p" | "q";
E: "st";
```

In this set of rules, A, B, C, and E are non-terminal symbols since they have associated RHS. "D", "p", "q", and "st" are terminal symbols since they do not appear on the LHS of any rule.

When the RHS of a rule comprises both "&" and "|" operators, by default, they are treated left-associatively. In other words, the elements are grouped from left to right. The precedence of these two operators is the same. The associativity may be overridden by enclosing within parentheses as needed.

For example, the rule

```
A: (B | C) & (D & E);
```

without the explicit parentheses will be treated as

```
A: (((B | C) & D) & E);
```

The RHS may optionally be annotated with repetition markers. A repetition marker indicates how many times the particular element needs to be repeated during the generation process. The following marker formats are supported:

- number => indicates that the element is to be repeated ***number*** times.
- number* => indicates that the element is to be repeated ***0 to number*** times.
- number+ => indicates that the element is to be repeated ***1 to number*** times.
- * => indicates that the element is to be repeated ***0 to infinity*** times.
- + => indicates that the element is to be repeated ***1 to infinity*** times.

Examples:

```
P : Q & R | T;
Q : (“r” | “s”)3+; // The RHS of Q will be repeated 1, 2, 3 times
R: (“abc”) 2* & S; // The string “abc” will be repeated 0, 1, 2 times
S: (“lkj” & “opu”)23; // RHS to be repeated 23 times
T : “zxcvb| +; // Repeat 1... times
```

Note

- When "*" or "+" is used, the generation process will not terminate on its own. The only way to terminate is to attach some pre/post action code to the node and handle termination within the action code.

The marker "1*" is equivalent to enclosing the RHS element within "[ ]". That is,

```
A: (B)1*; // Do B zero or once
```

and

```
A: [B]; // Do B zero or once
```

are equivalent.

The following is a complete (although trivial) input specification:

```
// Sample grammar
// File: sample.grm
grammar sample;
structure {
        start: A & B | C;
        A: "AA";
        B: "BB";
        C: "CC";
}
```

Every grammar must have a special non-terminal symbol called *start*. This is the starting point for generation.

ACTIONS

JSynTest reads the language definition file and internally represents the information as an AND-OR graph made up of terminal and non-terminal nodes. It then emits a Java program that implements the graph and which when executed, traverses the graph by visiting the constituent nodes and appending the generated sub strings to an output buffer. To support context-sensitive processing of the nodes, certain actions can be carried out as part of visiting a node. These actions are user-defined Java code fragments and are defined in the actions section of the grammar file. In order to facilitate reusing action code at multiple nodes, each action can be given a name, and bound to non-terminal nodes in the bindings section.

Both the actions and bindings sections are optional. When only the structure is defined, the default behavior is to send the strings generated to standard output.

For anything other than trivial string generation, actions will have to be attached to one or more non-terminal nodes in the graph. Associating an action code with a node is done in two steps. First the action code is named and secondly, it is bound to a node in the bindings section. This two-step process facilitates reuse of an action code at multiple nodes.

In the AND-OR graph, a node may be traversed many times depending on the structure of the graph. For example, if a node is an OR node, it may have many choice points and hence might be visited at least as many times. An action code can be bound to a node, to be called by the graph traversal algorithm at one of six points during the traversal. These are:

- **init:** The action code is executed only once just before traversal starts.
- **start:** The code is executed once at the beginning of each traversal of the graph.
- **pre:** The code is executed just before visiting the node.
- **post:** The code is executed just after visiting the node.
- **stop:** The code is executed at the end of each traversal of the graph.
- **destroy:** The code is executed just once when the graph traversal is complete.

The following example illustrates the various action code entry points.

```
grammar allactions;
structure {
      start: A & "is the result\n";
      A: B | C;
      B: "this-b ";
      C: "this-c ";
}
actions {
      Token.pre: %{
            System.out.println("About to enter Node");
            return LG_CONTINUE_NORMAL;
      %}
      // This will be called just once, at the very beginning.
      Token.init: %{
            System.out.println("Node object initialized");
      %}
      // This will be called just once, at the very end.
      Token.destroy: %{
            System.out.println("Node object destroyed");
      %}
      // This will be called once at the beginning of every traversal.
      Token.start: %{
            System.out.println("Traversal starts");
      %}
      // This will be called once at the end of every traversal.
      Token.stop: %{
            System.out.println("Traversal ends");
      %}
```

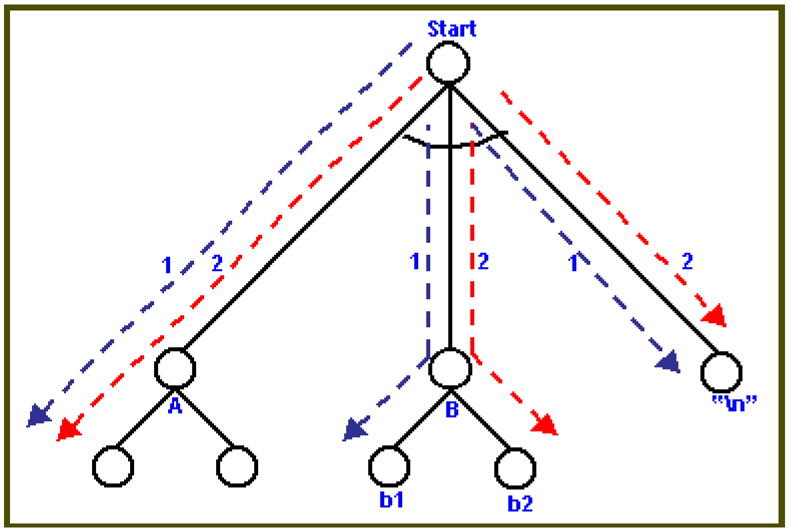
```
        // This will be called just after the node is visited.
                Token.post: %{
                System.out.println("Node has been visited");
                return LG_CONTINUE_NORMAL;
        %}
}
bindings {
        B: Token;
}
```

AND-OR Graph

As outlined earlier, at the heart of syntax testing is the construction and traversal of an AND-OR graph. The grammar that captures the input domain is internally represented as an AND-OR graph. At runtime, this graph is traversed appropriate number of times to synthesize all possible strings derivable from the grammar.

```
grammar simple;
structure {
start: A & B & "\n";
A: "a1" | "a2";
B: "b1" | "b2";
}
```

We can represent this grammar as the following AND-OR graph (in this case, an AND-OR tree):



The start node has an arc to suggest that it is an AND node. “A” and “B” are OR nodes. The other nodes are leaf nodes. An AND node is considered traversed only when all its children are traversed, whereas an OR node is considered traversed when any one of its children is traversed.

To traverse start, we need to traverse “A” and “B” and “\n”. Since “A” and “B” are OR nodes, taking any one of their children is sufficient for a traversal. This results in 2 * 2 * 1 combinations, or 4 combinations in all. Thus, the outputs emitted during the graph traversal are

```
a1    b1    \n
a1    b2    \n
a2    b1    \n
a2    b2    \n
```

To better understand **JSynTest** and its formalism, let us consider the following problem.

AND-OR Graph Representation

A Complete Example

Let us say we are interested in generating all valid dates in the range 1/1/2000 to 31/12/2001 (dd/mm/yyyy), both dates inclusive. This data set might be required as an input to an application that uses date fields, but we will ignore the modalities of how the data generated by the tool will be fed to the application.

Here is our first attempt at defining a grammar for valid dates.

```
grammar Date;
structure{
      start : month & "/" & day & "/" & year & "\n";
      day : "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "10" |
      "11" | "12" | "13" | "14" | "15" | "16" | "17" | "18" | "19" | "20" |
      "21" | "22" | "23" | "24" | "25" | "26" | "27" | "28" | "29" | "30" |
      "31";
      month : "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "10" |  "11" | "12";
      year : "2000" | "2001";
}
```

This grammar is self explanatory. After **JSynTest** parses the grammar, it will synthesize a Java program that builds an AND-OR graph corresponding to the grammar and traverses the graph as many times as needed to generate all possible combinations of output. We can easily see that the total number of combinations in this case will be 31 * 12 * 2.

It is worth remembering that the same output data can be generated by different grammars. The following grammar, for instance, generates the same set of strings as the grammar above.

```
grammar Date2;
structure {
        start : month & "/" & day & "/" & year & "\n";
        digit: "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
        digit2: "0" | digit;
        month: digit | "10" |  "11" | "12";
        day: digit | ("1" & digit2) | ("2" & digit2) | "30" | "31";
        year : "2000" | "2001";
}
```

A compact grammar that generates exactly the required data, without generating all possibilities and then eliminating some, is preferable to a grammar that generates superfluous data and subsequently filters them. Just as in programming and other development efforts, it takes patience and experimentation before we arrive at a good grammar.

One problem with the above grammars is that they also emit invalid dates, such as 31/2/2000, 31/4/2001, and so on. In some cases, of course, we would have to generate invalid dates, but let us ignore that requirement for now. To eliminate invalid data, we have to associate action code with some of the nodes. Here is the modified grammar:

```
grammar Date;
structure{
        start : month & "/" & day & "/" & year& "\n";
        day : "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "10" |
        "11" | "12" | "13" | "14" | "15" | "16" | "17" | "18" | "19" | "20" |
        "21" | "22" | "23" | "24" | "25" | "26" | "27" | "28" | "29" | "30" | "31";
        month : "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "10" | "11" | "12";
        year : "2000" | "2001";
}

actions {
        actYear.def: %{
        //month with 30 days
        String[] monWith30Days = {"4", "6", "9", "11"};
        //checks if month has 30 days
        boolean is30DayMonth(String mon){
                for(int i=0; i<monWith30Days.length; i++)
                if(monWith30Days[i].equals(mon))
                return true;
                return false;
        }
        //checks for leap year
```

```
            boolean isLeapYear(String yr) {
                  boolean leap;
                  int y = Integer.parseInt(yr);
                  if(y%100 ==0){
                        if(y%400 == 0) leap = true;
                        else leap = false;
                  }
            else {
                  if(y%4 == 0 ) leap = true;
                  else leap = false;
            }
                  return leap;
            }
      %}
      actYear.post: %{
            String dyStr = out.getStringAt(2);
            String mon = out.getStringAt(0);
            String yr = out.getGeneratedString(node);
            int dy = Integer.parseInt(dyStr);
            if(mon.equals("2")){
                  if(isLeapYear(yr)){
                        if(dy > 29)
                        return LG_MOVE_NEXT;
                        //skip nodes that have values greater than 29
                  }
                  else {
                        if(dy > 28)
                        return LG_MOVE_NEXT;
                        //skip nodes that have values greater than 28
                  }
            }
            else if(is30DayMonth(mon)){
                  if(dy > 30)
                  return LG_MOVE_NEXT; //skip nodes that have values greater than 30
                  }
            return LG_CONTINUE_NORMAL;
      %}
```

```
}


bindings {
        year: actYear;
}
```

In the above grammar, we have associated an action named **`actYear`** with a node named year. As mentioned earlier, the section **`actYear.def`** defines fields and methods that will be emitted as part of the Java class. The section **`actYear.post`** defines a Java method that is invoked every time after the year node is visited. Inside the post method, the variable out refers to the output buffer that accumulates the strings emitted till the end of each traversal. By default, the buffer contents are written to the standard output stream at the end of each traversal and the buffer is cleared.

Reuse of Grammars

As we started to design our grammatical formalism, it became apparent that grammars were potentially reusable entities. Taking inspiration from object orientation, we decided to support composition and derivation among grammars. This means we could develop grammars incrementally, building upon other grammars. As an example, if we are building a specification for a language such as Java, we could define a grammar for expressions, use that to build a grammar for statements, from there to methods, to classes, and so on. This is a compositional approach. Similarly, we could define base grammars common to a family of languages, and then specialize these to apply to specific languages.

The date example discussed above involves three fields - day, month and year. We can think of two other examples that involve three fields: telephone number and social security number. Is there a way to capture the commonalities (and differences) between these three examples?

First, we define an abstract grammar that defines the top-level structure of these data:

```
abstract grammar ThreeFieldGrammar;
structure {
        start: field1 & separator & field2 & separator & field3 & "\n";
        separator: "-";
        abstract field1;
        abstract field2;
        abstract field3;
}
```

An abstract grammar captures an incomplete specification. A non-terminal node must be declared *abstract* if it does not have a corresponding RHS. A grammar that has at least one abstract node must be declared *abstract*. A derived grammar typically provides the definition (that is, RHS) for the abstract node. In case the derived grammar does not provide the definition for any inherited abstract node, then the grammar must be declared abstract. An abstract grammar that defines one or more abstract nodes is a template grammar that defines the overall structure of the target language, where the specifics are supplied by a derived grammar.

In the above example, the non-terminals “field1”, “field2” and “field3” are left undefined. We then define a date grammar that describes what “day”, “month” and “year” are without actually specifying how they should be combined. For simplicity, we will use a modified form of the first version of the date grammar discussed earlier (without actions).

```
abstract grammar Date;
structure{
      day : "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "10" |
      "11" | "12" | "13" | "14" | "15" | "16" | "17" | "18" | "19" | "20" |
      "21" | "22" | "23" | "24" | "25" | "26" | "27" | "28" | "29" | "30" |
      "31";
      month : "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "10" |  "11" | "12";
      year : "2000" | "2001";
}
```

We could similarly define grammars for telephone number and social security number, each of which describes three fields without suggesting how they are combined.

Given these, the following grammar specifies a concrete three-field date:

```
grammar  DDMMYYGrammar extends ThreeFieldGrammar;
structure {
      field1: Date.day; // composition
      field2: Date.month; // composition
      field3: Date.year; // composition
}
```

Notice how we derive from `ThreeFieldGrammar` and compose from `Date`. If we want to generate dates in MM/DD/YYYY format, the following grammar does the job:

```
grammar  MMDDYYGrammar extends ThreeFieldGrammar;
structure {
      field1: Date.month;
      field2: Date.day;
      field3: Date.year;
}
```

A similar strategy is applicable to telephone number and social security. For example,

```
grammar TelephoneGrammar extends ThreeFieldGrammar;
structure {
      field1: TelephoneNumber.areacode;
```

```
        field2: TelphoneNumber.number1;
        field3: TelephoneNumber.number2;
}
```

As in the case of object orientation, derivation and composition of grammars prove useful ideas in syntax testing.

Final Grammar

A final grammar is one that cannot be derived from. A grammar that has no scope for refinement may be qualified as *final*.

```
final grammar simple;
structure {
        start: "A" & "B" | "C";
}
```

It is also possible to declare a node as *final*, implying that the node cannot be overridden in a derived grammar.

Java Integration

JSynTest is a Java application and hence can be used on any platform where Java is supported (we have tested the application on **Windows™**, **RedHat Linux**, and **Solaris 8™**, all on Intel PCs). In addition, since the synthesizer emitted by **JSynTest** is a Java program, the program along with the supplied runtime library can similarly be used on any platform. If necessary, the emitted source can be modified and embedded in another Java application.

Where has Been JSynTest Applied?

JSynTest is a new product from our stable and we are trying to find interesting applications for the tool. We have so far applied the tool in the following domains:

- Generating Intel 8085 CPU assembly language instructions.
- Generating JVM instruction set that can be compiled into a **.class** file (this was part of an experiment to check JVM security issues).

Other Applications

Although **JSynTest** is intended primarily for use in syntax testing, given that the core logic involves building and traversing an AND-OR graph, it is possible to use the tool in problems that benefit from a "generate-and-test" paradigm. One such problem is the 8-Queen puzzle that requires placement of eight queens on the chessboard such that no queen is in the path of another. Here is the grammar with action code (full source is not included):

```
grammar Q8;
structure{
      start: startline & "\r\n";
      startline: A & A & A & A & A & A & A & A;
      pos : "0" | "1" | "2" | "3" | "4" |"5" |"6" |"7";
      A: pos;
}

actions {
      QLogic.post: %{
      LGNamedNode named = node;
      LGNode y = named.getGrammarObject().getNamedNode("pos");
      int ypos = Integer.parseInt(out.getGeneratedString(y));
      if(logic.addQueen(ypos) == false){
            return LG_MOVE_NEXT;
      }
            return 0;
      %}
       QLogic.def: %{
            QLogic logic = new QLogic();
       %}
       QLogic.start:%{
            logic.reset();
       %}
}

bindings{
      A:QLogic;
}
```

Conclusion

Syntax testing is a powerful black box testing strategy for testing an application whose input domain can be characterized by a grammar. Many real world applications can benefit by this technique. It is also a technique that can most readily be automated. **JSynTest** is a GUI-based framework for syntax-based testing.

Further Reading

1. Boris Beizer, Software Testing Techniques, 2nd Edition, Van Nostrand Reinhold, 1990.
2. Boris Beizer, Black-Box Testing, John Wiley & Sons, 1995.