yagg: An Easy-To-Use Generator for Structured Test Inputs

David Coppit
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185
david@coppit.org

Jiexin Lian
Department of Computer Science
The University of Illinois
Chicago, IL 60607
jlian2@uic.edu

ABSTRACT

Automated testing typically uses specifications to drive the generation of test inputs and/or the checking of program outputs. Many software systems have structurally complex inputs that cannot be adequately described using simple formalisms such as context-free grammars. In order to generate such inputs, many automated testing environments require the user to express the structure of the input using an unfamiliar formal notation. This raises the cost of employing automated testing, thereby offsetting the benefits gained. We present *yagg* (yet another generator-generator), a tool that allows the programmer to specify the input using a syntax very similar to that of LEX and YACC, widely used scanner and parser generators. yagg allows the user to bound the input space using several different techniques, and generates an input generator that systematically enumerates inputs. We evaluate the ease of use and performance of the tool relative to a model checker-based generator used in previous research. Our experiences indicate that yagg generators can be somewhat slower, but that the ease-of-use afforded by the familiar syntax may be attractive to users.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Algorithms,Verification

Keywords

bounded exhaustive testing, grammar-based input generation

1. INTRODUCTION

Automated input generation is a key component of many testing strategies. For example, for both random and exhaustive testing, inputs are automatically created from some formal description of the input space, the system is executed for those inputs, and the results checked for correctness. The intuition behind such approaches is that they can reveal failures involving uncommon inputs, beyond those that are revealed by other test selection techniques. Embedded software, for instance, can be tested against random interrupt schedules to help reveal subtle timing faults.

For many systems, the generation of valid inputs is a nontrivial task due to their structural complexity. This difficulty was first recognized during the development of the first compilers for high-level programming languages. Early approaches were based on context-free grammars, and were unable to represent context-sensitive structural constraints, or were difficult to use. More recent approaches are based on specification languages. The developer must create an input specification in an unfamiliar declarative language, which can lead to incorrect specification of the input, and potentially less rigorous testing. During testing, the developer may also modify the input specification in order to focus the testing process on particular regions of the input space. Making such modifications in an unfamiliar language increases the chance of error. Lastly, many tools that generate inputs focus on random generation, avoiding the difficulties of exhaustive generation.

In this paper, we present *yagg* (yet another generator-generator), a tool that takes as input a specification of the input space, and generates a generator that is able to enumerate all the inputs of a user-specified length. A key advantage of this tool is that its input is in a format that is very close to the widely used LEX [7] and YACC [6] tools. Many programmers are already familiar with the syntax of the specification files for these tools. Further, it is not uncommon for software to already have LEX and YACC input files that can be used with little modification, further lowering the cost of using yagg to generate inputs for testing.

To evaluate the usability and performance of the tool, we compare it against an input generator developed by collaborators in previous work [2]. We use both tools to generate dynamic fault trees (DFTs), structurally complex models used by reliability engineers. Our experiences suggest that yagg can be easier for programmers to use, but that the resulting generators produce inputs at a somewhat slower rate than the previous tool, at least on machines with large amounts of memory.

In the next section we describe related work. Section 3 describes the key features of yagg. In Section 4, we evaluate the usability and performance of the tool, and Section 5 concludes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASE'05, November 7–11, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-58113-993-4/05/0011 ...$5.00.

2. RELATED WORK

2.1 Parser Testing

Input generation has been employed by compiler developers to test parsers for correctness, both in terms of parsing of correct programs and rejection of incorrect programs. Purdom used context-free grammars to achieve "production coverage," arguing that ex-

```
%{
extern int yylex();
extern void yyerror(char *s);
double result;
unsigned int number_of_operators = 0;
%}

%union {
  double number;
  int    token;
}

%type  <number> exp
%token <number> NUMBER
%left           PLUS
%left           DIVIDE
%token          NEWLINE

%%

exp_line :
  exp NEWLINE {
    if (number_of_operators % 2 != 0)
      yyerror("Only even number of operators!");

    result = $1;
  } ;

exp :
  exp PLUS exp {
    $$ = $1 + $3;
    number_of_operators++;
  }{ number_of_operators--; } |
  exp DIVIDE exp {
    if ( $3 == 0 ) {
      yyerror("Can't divide by zero");
      $$ = $1;
    } else
      $$ = $1 / $3;

    number_of_operators++;
  }{ number_of_operators--; } |
  NUMBER ;
```

Listing 1: Grammar file for a context-sensitive arithmetic expression generator

```
%{
#include <iostream>
//#include "parser.tab.h"
%}

%%

"\n" return NEWLINE;
"+" return PLUS;
"/" return DIVIDE;

( 1.2 | 0 | 5 ) return NUMBER;
/*
[0-9]+.[0-9]+ {
  yylval.number = atof(yytext);
  return NUMBER;
}
*/

%%

int yywrap() {
  return 1;
}

void yyerror(char* error_string) {
  std::cerr << error_string << std::endl;
}
```

Listing 2: Terminal file for a context-sensitive arithmetic expression generator

ercising the language grammar suffices for testing the parser [11]. Later work [1, 3], used attributed context free grammars (CFGs) to encode context-sensitive constraints, as well as the expected output. Using this approach, full test cases involving syntactically valid programs can be generated. The authors are unware of any publicly available input generator tools based on attributed CFGs.

2.2 Alternative Tools

Simple CFG-based random input generation is relatively trivial to implement. Maurer's data generation language (DGL) [9] is a robust tool that is similar in spirit to yagg. DGL supports grammar constructs such as "actions" and "chains" for including context-sensitive elements and enabling exhaustive generation. However, restrictions on their use limit their utility. For example, actions cannot be used to invalidate a generated string, and the chain feature can only be used for exhaustive enumeration if the grammar is weakened from a general context-free grammar to one obeying a more restrictive form. These and other limitations prevented us from being able to use DGL to generate dynamic fault trees. As a result, this tool is not included in our evaluation.

In previous work, we developed a systematic input generator based on the TestEra [8] tool, as part of an evaluation of bounded exhaustive testing [2]. TestEra input descriptions are written in Alloy [4], a first-order declarative specification language whose semantics is based on relations. We used TestEra in a two-stage generation process to efficiently and exhaustively generate all inputs at user-specified scopes.

yagg is distinct from these existing tools in its explicit goal of reusing the syntax of LEX and YACC to the maximum extent possible. For this reason, yagg has the potential to be easier to use than previous tools. For example, TestEra input specifications are written in Alloy, and the tool requires some expertise for the development of symmetry-breaking total orders. yagg uses imperative programming to enforce context-sensitive constraints on the generated inputs. While this approach may be more familiar to programmers, TestEra can possibly generate inputs more quickly, due to its underlying use of state-of-the-art SAT solvers [5]. For inputs having simple structure, DGL may be useful, but requires the programmer to learn a difficult-to-use new language.

3. USAGE AND FEATURES

3.1 Overview

yagg is an input generator generator. Given YACC-like and LEX-like input files, yagg generates a C++ program that generates all strings of a user-specified length. The generated generator constructs the syntax tree for the specified length, then enumerates all possible strings for that tree. The YACC-like language grammar file defines the context-free structure of the input space, and contains action blocks that validate generated candidate inputs with respect to context-sensitive checks. The LEX-like terminal generator file provides specifications that instruct the program how to generate strings for terminals in the grammar.

3.2 Grammar Files

The yagg grammar file format is the same as that of YACC, with one exception. Because yagg generates inputs in an iterative manner, any grammar production whose action block has side effects must also provide another "unaction block". These are used during generation to restore any state that was changed as a result of executing the action block for the rule. yagg automatically restores changes to `$$` or any of the `$1`, `$2`, etc. positional parameters, so programmers usually only need to add unaction blocks when an action block modifies global variables.

Within an action block, programmers typically check context sensitive constraints, calling `yyerror(char*)` when one is violated. yagg automatically detects such a call, and suppresses the generation of the string. Also, because yagg executes action blocks many times, the programmer must properly release dynamically allocated memory in order to avoid increasing memory usage.

3.3 Terminal Files

yagg's terminal file format is similar to that of LEX. Typically, a LEX scanner specification uses regular expressions to recognize a wide range of values for particular tokens. However, generating all possible values for a regular expression is generally not useful, since such strings generally exhibit a large amount of isomorphism. Instead, yagg provides a syntax that allows the user to specify one of a number of different strategies for generating a more limited set of values for terminals. Instead, yagg supports several *terminal specifications*. A *simple* expression such as `"="` represents a single constant string. An *alternation* expressions such as `("+"|"-"|"*"|"/")` represents alternatives, each of which is enumerated during string generation.

Equivalence alternation expressions of the form `["x"|"y"]` form an equivalence class with respect to terminal names (but not identities). For example, consider the production

```
sum :  VARIABLE "+" VARIABLE ;
```

and the terminal specification

```
["x"|"y"] return VARIABLE;.
```

In this case, the strings "x+x" and "x+y" would be generated, but not "y+x" or "y+y." This terminal specification type is useful for identifiers, whose names are only needed to distinguish different variables, but are not important otherwise.

Equivalence generator expressions are semantically the same as equivalence alternation, except that the strings are generated on-the-fly as needed. For example, `["var_#"]` will generate strings of the form "var_1", "var_2", etc. as required by the grammar.

3.4 Example

Listings 1 and 2 show the grammar and terminal specification files for a simple expression generator. In this example, context-sensitive checks are added to ensure that the expression has an even number of operators, and that no division by zero occurs (within the accuracy of floating point arithmetic). Although yagg supports a more terse input format that merges and reduces these two files, we have provided these input files to demonstrate that they can be easily derived from YACC and LEX files.

The `exp` nonterminal demonstrates the use of unaction blocks to decrement the `number_of_operations` variable as part of the restoration of state during generation. Technically the `result` in the `exp_line` nonterminal should also be restored, but this is unnecessary as no grammar rule depends upon this global state. This grammar file is nearly identical to the input to YACC, with the exception of the unaction blocks.

The terminal file demonstrates the use of a simple alternation for the generated number, which replaces the use of a regular expression in the original LEX input file. Other than this change, the only other change to the LEX input file was to disable the inclusion of the parser header file, which is not needed by yagg.

This example demonstrates the ease of use of the yagg tool. Many developers are already familiar with the syntax of the standard LEX and YACC scanner and parser generators, and may already have developed such input files for the software under test. yagg provides useful ways to shape the generated input space through the choice of terminal specification type, alteration of the input grammar, or the use of context-sensitive checks.

4. EVALUATION

In previous work, we evaluated the feasibility of bounded exhaustive testing for nontrivial systems having structurally complex inputs [2]. A substantial part of that research effort was devoted to the development of efficient techniques for input generation. We used this state-of-the-art input generation system as our benchmark, evaluating yagg against this system in terms of usability and performance. The input domain we generated is that of dynamic fault trees (DFTs) [12]. DFTs are essentially a language for modeling the reliability of complex systems, where the engineer combines basic events, gates, and constraints to build a model of the system. The DFT language has a nontrivial syntax that precludes the generation of DFTs using simple context-free grammars.

4.1 Usability

In our previous work, we used the Alloy specification language to specify the inputs to be generated, as part of a two-phase generation process in which TestEra generates abstract inputs which are then combinatorially instantiated by a hand-written postprocessor. Alloy is a declarative specification language based fundamentally on relations. As a result, the DFT model is quite concise, at only 127 lines of code. However, developing the model was not trivial, and required the expertise of a TestEra developer. This was particularly important for the development of symmetry-breaking predicates that prevent the generation of isomorphic inputs.

Compared to the two-phase generation approach, yagg has several advantages. First, it is fully automated, requiring no development of post-processor programs. Second, the start-up cost was quite low. Since we were already familiar with writing parser input files, we did not need to become adept in Alloy. Indeed, we already had parser and scanner input files for YACC and LEX. We simply modified the parser file to include unaction blocks, and replaced the scanner file with appropriate terminal specifications.

There were some complications with the use of yagg to generate DFTs. Because the context-sensitive aspects of the grammar are imperative in nature, the fault tree grammar file was substantially larger than the Alloy model, at 1524 lines. The connectedness constraint, for instance, is four lines of Alloy specification, but 112 lines of C++ code. While programmers may be more comfortable with an imperative style, using it may require more work. We also had to be careful about memory leaks in the action blocks, since a memory leak will be magnified by the repeated execution of action blocks during input generation.

In both tools, extra constraints had to be added to avoid generating redundant isomorphic inputs. In yagg, this entailed both modifying the grammar and adding more checks within the action blocks. For example, we modified the grammar to ensure that the different gate types appear in a canonical order. We also modified the input sequence action for gates to ensure that the inputs to order-independent gates are in a canonical alphabetical order. One could argue that performing these modifications are more complicated than adding additional predicates in the Alloy model.

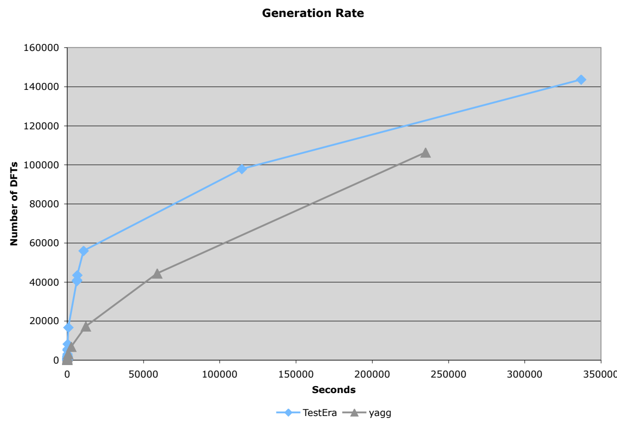

Figure 1: A performance comparison of yagg and TestEra+postprocessor

4.2 Performance

In order to assess the performance of yagg-generated input generators, we compared the performance of the DFT generator created by yagg against our previous TestEra-based generator. We ran each generator over a span of several days on a 3 GHz Linux PC having 512 KB of cache and 2 GB of main memory.

Figure 1 summarizes the number of fault trees that were generated over time. The two-phase TestEra-based generator outperformed the yagg generator, although the gap may be closing. We hypothesize that the TestEra-based solver is faster due to its symbolic manipulation of the input space to find instances that satisfy the constraints. In contrast, yagg uses the context-free grammar to generate many candidate inputs, which are then validated by the context-sensitive checks in the action blocks. In some cases, it is possible that yagg's generate-then-filter approach may be substantially slower than TestEra's symbolic-manipulation-then-generation approach. (It should be noted that the underlying SAT solver, mChaff [10], is not optimized for enumeration.)

In terms of memory performance, the yagg generator is dramatically better. TestEra's underlying SAT solver consumes large amounts of memory—over 1 GB toward the end of the experiment. In contrast, the yagg-based tool consumes less than 3 MB of memory at all times. On machines with less memory, we expect the yagg generator to significantly outperform the TestEra-based one as the latter swaps memory to disk. It is also possible that the yagg generator will outpace the TestEra-based one in the long run as the SAT solver consumes increasingly more memory for larger inputs.

5. CONCLUSION

In this paper we have presented yagg, a tool for the generation of exhaustive input generators. yagg has an interface that is familiar to many programmers, and yet the tool is powerful enough to generate inputs for complex structures such as dynamic fault trees. Although the CPU-bound performance is not as good as that of our previous benchmark generator, its memory consumption is much better. When a YACC input file is already available, and the programmer is familiar with its syntax, yagg provides a very low-cost method for generating inputs for exhaustive testing.

In the future, we hope to improve the performance by employing code analysis to improve yagg's rather conservative caching of generated strings. We also plan to add support for random generation of inputs. An interesting problem in this dimension is how to remove the bias in the randomly generated strings that results from the structure of the grammar. We also recognize the potential difficulty of writing correct unaction blocks, and hope to perform code analysis so that we can automatically save and restore the state so that the user need not write them. Finally, in principle yagg can be used to generate programs for compilers, but we have not yet had the opportunity to test it for large languages such as C++.

Acknowledgments

The authors would like to thank Kevin Sullivan, Jinlin Yang, and Wei Le for their work on the TestEra-based generator, and for providing access to that tool for our evaluation.

6. REFERENCES

[1] Jonathan A. Bauer and Alan B. Finger. Test plan generation using formal grammars. In *Proceedings of the 4th International Conference on Software Engineering*, pages 425–32, Munich, Germany, 17–19 September 1979. IEEE.

[2] David Coppit, Jinlin Yang, Sarfraz Khurshid, Wei Le, and Kevin Sullivan. Software assurance by bounded exhaustive testing. *IEEE Transactions on Software Engineering*, 31(4):328–39, April 2005.

[3] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *Proceedings of the 5th International Conference on Software Engineering*, pages 170–8, New York, NY, 9–12 March 1981. IEEE.

[4] Daniel Jackson. Micromodels of software: Modelling and analysis with Alloy. URL: `http://sdg.lcs.mit.edu/alloy/reference-manual.pdf`.

[5] Daniel Jackson, I. Scheckter, and I. Shlyakhter. Alcoa: The Alloy constraint analyzer. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 730–3, Limerick, Ireland, 4–11 June 2000. IEEE.

[6] S. C. Johnson. YACC — Yet another compiler-compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, N.J., 1975.

[7] M. E. Lesk and E. Schmidt. Lex — A lexical analyzer generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, N.J., 1975.

[8] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering (ASE 2001)*, San Diego, CA, 26–29 November 2001. IEEE.

[9] Peter M. Maurer. The design and implementation of a grammar-based data generator. *Software Practice and Experience*, 22(3):223–44, March 1992.

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–5, Las Vegas, Nevada, 18–22 June 2001.

[11] P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3):366–75, 1972.

[12] W. E. Vesely, Joanne Dugan, Joseph Fragola, Joseph Minarick III, and Jan Railsback. *Fault Tree Handbook with Aerospace Applications*. National Aeronatics and Space Administration, August 2002.