

# yagg: Implementation and Evaluation

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

## ABSTRACT

Automated testing seeks to reduce the cost of verification, typically by using 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 describe the features of the tool, as well as its design and implementation. 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, both random and exhaustive testing can be employed at the system level, where 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 ensure the absence of subtle timing faults.

For many systems, the generation of inputs is a nontrivial task. The set of all possible inputs to the program can be partitioned into valid and invalid inputs. Testers are interested in generating both classes of inputs: valid inputs for the correctness of the software, and invalid inputs for the robustness of the software. For systems having structurally complex inputs, the set of valid inputs is relatively small, and more difficult to generate. The developer must provide the input generator with a formal description of the valid input space, which the generator then uses to perform the generation. In general, the input specification language must be sufficiently powerful to document context-sensitive constraints in the structure of the input.

Generating structurally complex inputs is an old problem, dating back to the development of the first compilers for high-level programming languages. Purdom [22] presented a simple, efficient algorithm for enumerating strings from a context-free grammar. Hanford [12] presented a fairly complex technique for also handling context-dependent constraints using dynamically altered grammars. Later researchers explored alternative strategies for representing context-sensitive dependencies, using formalisms such as attributed context-free grammars [2, 11] and context-free parametric grammars [3]. More recent efforts have focused on the use of specification languages such as Z [13], UML [21], JML [4], and Alloy [18].

Early approaches were unable to represent context-sensitive structural constraints, or were difficult to use. More recent approaches require the developer to learn a new formalism with which to specify the input space. The developer must create an input specification in this unfamiliar 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 modification 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 [17] and YACC [16] tools. Many programmers are already familiar with the

```

%%
exp : exp binary_operator exp | number ;

binary_operator : "+" | "-" | "*" | "/" ;

number : "1.2" | "0" | "5" ;

```

**Listing 1: A simple arithmetic expression example**

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.

yagg supports a number of different mechanisms that allow the programmer to control the scope of inputs that are generated. The code that it generates is object-oriented C++, which is designed to accommodate multiple generation strategies, and to generate inputs in an efficient manner. At the same time, yagg goes to great lengths to use existing LEX and YACC input files unchanged.

To evaluate the usability and performance of the tool, we compare it against an input generator developed by collaborators in previous work [7, 23]. Our previous generator is based on TestEra [18], which in turn uses a model checker called the Alloy analyzer [15]. 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 TestEra-based 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. Section 4 describes the design and implementation of the tool. In Section 5, we evaluate the usability and performance of the tool, and Section 6 concludes.

## 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 exercising the language grammar suffices for testing the parser [22]. Later work [2, 11], 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.

Research in the generation of parser test data seems to have slowed in recent years. This may be due to the advent of robust tools for automatically generating parsers, such as YACC [16]. Another cause might be the development of standard compiler test suites. The authors are unaware 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. As we will argue in Section 4, both context-sensitive constraints and exhaustive enumeration significantly complicate the creation of a more useful input generator. Maurer’s data generation language (DGL) [19] is a robust tool that is similar in spirit to yagg. Using DGL, the user writes a grammar that describes the input.

GNUmakefile	← GNU make makefile
util/	
...	← Build helper programs
src/	
generator/	← Core generator framework
...	
model/	← Generator-specific files
nonterminal_rules/	← Nonterminals
...	
terminal_rules/	← Terminals
...	
utility/	← Shared functions
...	
progs/	
generate.cc	← The generator program

**Figure 1: A summary of generated files**

DGL supports 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, forcing the user to instead construct the valid string in a context-sensitive way. Similarly, the chain feature allows for exhaustive enumeration, but only 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.

Dascript by Godmar Back allows testers to generate binary data [1]. Using the dascript language, the programmer defines the structure of binary data as types. The dascript compiler then generates Java library code for reading and writing the binary format. In addition, the programmer can provide constraints on the types that are used to validate data that passes through the library.

In previous work, we developed a systematic input generator based on the TestEra tool, as part of an evaluation of bounded exhaustive testing [7, 23]. TestEra input descriptions are written in Alloy [14], a first-order declarative specification language whose semantics is based on relations. We modified TestEra to exhaustively generate all inputs at specified scopes, taking advantage of its automated and user-defined symmetry breaking capabilities to avoid generating isomorphic inputs. We also dramatically increased the efficiency of the input generation by splitting generation concerns, using TestEra to only generate the abstract structure of the input, and a manually developed postprocessor to combinatorially instantiate the abstract structure for concrete values.

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 [15]. For inputs having simple structure, DGL may be useful, but both DGL and Dascript require the programmer to learn a new language.

## 3. USAGE AND FEATURES

In this section we introduce the yagg tool. We present some of its features, and demonstrate its usage through two examples. The first is an example involves a minimal, simple grammar for the tool.

```

%{
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 MINUS
%left TIMES 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 {
        $$ = $1 + $3;
        number_of_operators++;
    } { number_of_operators--; } |
    exp MINUS exp {
        $$ = $1 - $3;
        number_of_operators++;
    } { number_of_operators--; } |
    exp TIMES 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 2: Grammar file for a context-sensitive arithmetic expression generator**

The second is a more complicated example that involves context-sensitive constraints, and strives to be compatible with YACC.

### 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

```

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

%%

"\n" return NEWLINE;
"+" return PLUS;
"-" return MINUS;
"*" return TIMES;
"/" 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 3: Terminal file for a context-sensitive arithmetic expression generator**

grammar. 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. 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.

Listing 1 shows a simple grammar file that defines an input space of arithmetic expressions. In this example, the terminals have been specified as part of the grammar, so no associated terminal specification file is needed.

## 3.2 Features

### 3.2.1 Invocation

Given an input description, yagg produces C++ code as summarized in Figure 1. The tool generates a makefile and a directory of utility programs for building the code. The `src` directory contains common generator code, code specific to the input model, and the main generator program. Using the input grammar and terminal files, yagg creates implementation files for each of the nonterminals and terminals, as well as any additional utility code. Using the included makefile, the user can compile this code, then run `progs/generate` with an argument specifying the length of the strings to be generated. yagg also allows the user to provide auxiliary implementation files for domain-specific data structures, algorithms, etc. It also has the ability to build and run the generator on a more powerful remote computer.

### 3.2.2 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.

### 3.2.3 Terminal Files

`yagg`'s terminal file format is similar to that of LEX. However, `yagg` does not attempt to generate all possible strings for a regular expression, since such strings generally exhibit a large amount of isomorphism. 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"]` treat the terminals as part of an equivalence class with respect to their names (but not their 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.

## 3.3 Example

Listings 2 and 3 show the grammar and terminal specification files for a more complicated example. 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). Unlike the previous example, the input files are based on those for a YACC-based parser and scanner.

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

1. Initialize program
2. Parse grammar and terminal input files
3. Create generator code locally
  - 3.1. Copy core generator files
  - 3.2. Generate code for terminals
  - 3.3. Generate code for nonterminals
  - 3.4. Generate utility code
  - 3.5. Generate main program
4. Copy new or modified files into output directory
5. Build generator code (optional)
6. Run generator (optional)

Algorithm 1: Overview of `yagg` generation algorithm

through the choice of terminal specification type, alteration of the input grammar, or the use of context-sensitive checks.

## 4. DESIGN AND IMPLEMENTATION

Simple context-free input generators for specific grammars are relatively easy to implement, and skilled developers often create such generators during the course of their careers. However, exhaustive enumeration, support for context-sensitive constraints, and the ability to generate arbitrary generators all significantly complicate the implementation of tools such as `yagg`. In this section we describe the high-level operation of `yagg` and provide an overview of the design of the generated code. We then describe the implementation in more detail with respect to exhaustive enumeration and context-sensitive constraints.

### 4.1 High-Level Design

`yagg` is an open source software tool written in Perl [29]. The total implementation consists of approximately 5,000 lines of code, not including the approximately 5,000 lines of C++ and template code used to create the generators. `yagg` is implemented using a number of Perl modules, the most important of which are `Parse::Yapp` [8] and `Text::Template` [9]. The former is used to create parsers for the grammar and terminal input files, and the latter is used in the generation of the generator code.

Algorithm 1 provides an overview of the `yagg` generation algorithm. During initialization, the program processes options and arguments, checking them for correctness. In order to parse the grammar and terminal inputs files, the tool uses a parser generated by `Parse::Yapp` from parser description files we developed. We developed the grammar description file based on that of Bison 1.05 [24], extending it to support unaction blocks. Similarly, the terminal grammar description file is based on that of Flex 2.54 [25], but modified to support our terminal specifications instead of regular expressions. (Bison and Flex are parser and lexer generators very similar to YACC and LEX.)

After initial parsing of the input files, some post-processing is done to determine the default starting rule, identify the nonterminals and terminals, extract shared code, and more. Virtual terminals are created for constant strings used in the grammar file, as was used in Listing 1. `yagg` also computes the minimal lengths of strings generated by each grammar rule. These lengths are used during generation as an optimization that avoids the construction of grammar subtrees for rules whose length is too short to result in any generated string.<sup>1</sup> Finally, `yagg` infers the types of terminal values. For example, Listing 1 used strings for the `NUMBER` terminal, whereas Listing 3 used unsigned doubles.

<sup>1</sup>Each grammar production can produce only 0 or more terminal strings, and each terminal can be an empty or nonempty string.

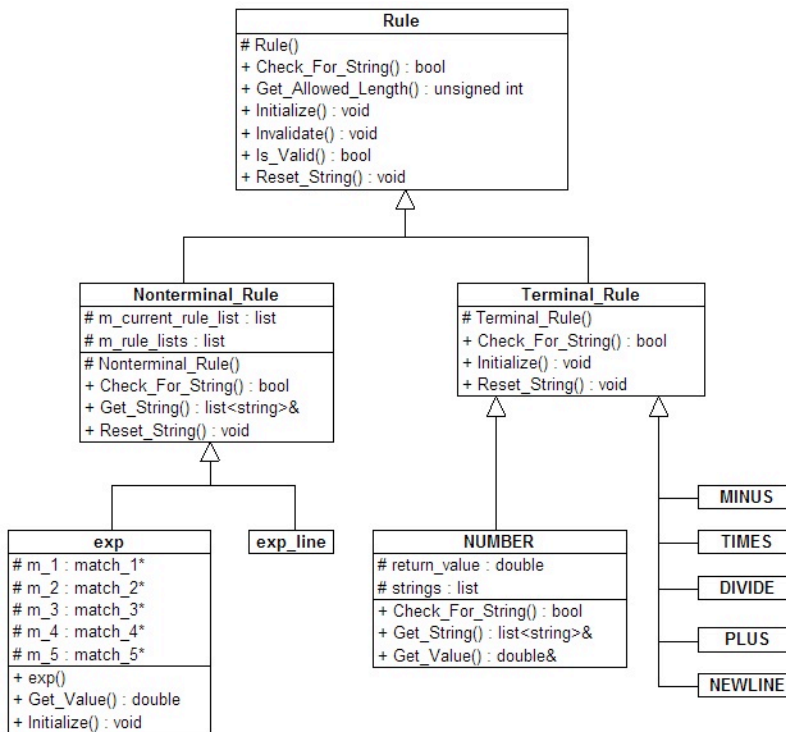


Figure 2: Class diagram of generated code

Once the parsing is complete, yagg generates the code. This is first done in a local temporary directory, then mirrored to the output directory using `rsync` [27]. This approach has two benefits. First, it avoids changing the modification time of unchanged files so that recompilation is faster. Second, for remote output, it avoids the performance overhead of writing many files over the network.

There are several steps in the generation of the C++ code. First, the tool copies files that are common to all generators. These include the makefile, base classes for the rules, and helper code. Next, it generates header and implementation files for each terminal, using one of four template files, depending on which terminal specification type the user specified in the terminal input file. For example, the equivalence generator template is used for any terminal whose specification is of the form `["<text>#<text>"]`. For each generated nonterminal, the tool checks the rule and any of its dependent rules for the presence of an action block or an equivalence terminal. The presence of either indicates that the rule may be context sensitive, in which case the generator cannot cache the generated strings. Depending on whether caching is possible, yagg generates a header and implementation file for each nonterminal.

Both the grammar and terminal input files may contain additional user-supplied code. This code is automatically extracted by yagg and stored in utility files. As part of this process, the tool automatically creates declarations for the header file, and moves definitions to the implementation file. In addition, the tool identifies any existing implementation of `yerror(char*)` and replaces it with a custom version that suppresses any error output, and notifies the generation framework not to continue generating the current string.

The main program is generated based on all of the rules specified in the input files. By default, the generator generates strings using the starting rule. However, the user can override this by specifying the name of a rule as an argument to the program. The generator program also checks the supplied string length to ensure that the

value is not smaller than the minimum length for the specified rule.

Throughout the generation process, `#line` directives are used to help the user identify compile errors in the original source files. The final two steps automatically build the code using GNU `make` [26], and run the generator for a specified length, provided the required options are given to yagg. Using a command such as `yagg -o userid@remote.example.com:output -r 5 math.yg math.lg`, the user can tell yagg to generate the generator program, then compile and run it for strings of length 5 on a remote computer.

## 4.2 The Generated Code

Figure 2 shows a class diagram of the generated rule code for the input files in Listing 2 and Listing 3. In the diagram, we have only provided the class information for one nonterminal and one terminal, in order to save space.

All generators share the `Rule`, `Nonterminal_Rule`, and `Terminal_Rule` classes in common. The latter two are derived from the first, and all rules specific to the user-supplied model are derived from either the `Nonterminal_Rule` or `Terminal_Rule` class. The `Rule` class defines an abstract interface of three methods that can be used to generate strings. The `Initialize()` method takes two arguments: a string length, and a pointer to the previous rule (whose default is `NULL`). This method resets the internal state of the rule, constructing the grammar tree for the given length. The `Check_For_String()` method takes no arguments. It increments the state of the grammar tree to the next string, checks that the string satisfies all context-sensitive constraints, and returns a Boolean indicating whether a valid next string exists. The `Get_String()` method takes no arguments and returns a list of strings for the current state of the rules.

The primary difference between nonterminal and terminal rules is that the former contains one or more `Rule_List` objects, each of

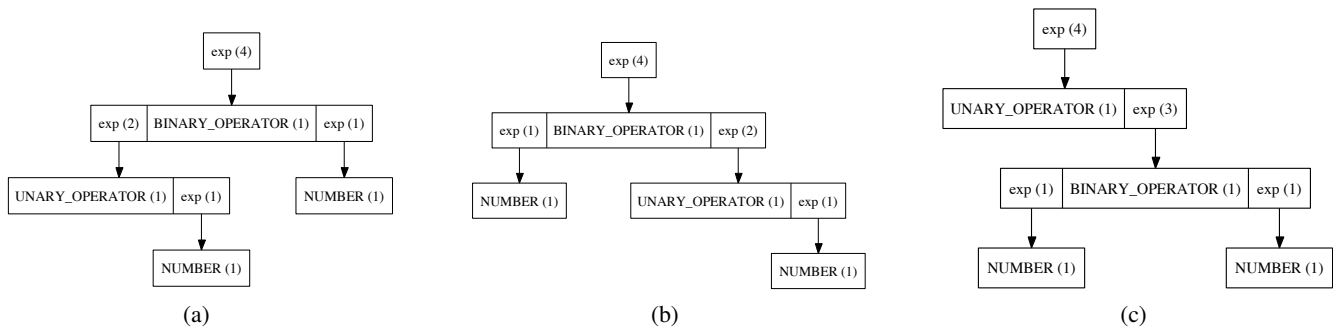


Figure 3: Different allocations for a grammar

which corresponds to a production in the original grammar. Rule lists contain references to other rules. Referring back to Listing 2, the `exp` nonterminal has five rule lists, corresponding to the four operator and the single number alternatives. These rule lists are private members of the `exp` class in the implementation, as shown in Figure 2.

Each rule list (class not shown) in a nonterminal rule derives from the `Rule_List` base class, overriding the `Do_Action()` and `Undo_Action()` methods, which are automatically generated from the action unaction blocks of the production in the grammar. During generation of these methods, position parameters such as `$1` are automatically converted by `yagg` into calls to the `Get_Value()` method for the appropriate rule.

### 4.3 Systematic Enumeration

The input generator systematically generates valid strings in two steps. In the first step, the generator attempts to allocate the user-specified length for the rule to be generated. (The length is the number of terminal symbols that appear in the final string, which may include empty symbols.) For example, consider the grammar production:

```
exp : exp BINARY_OPERATOR exp |
    UNARY_OPERATOR exp |
    NUMBER.
```

Given a length of 4, the generator attempts to construct a grammar tree by distributing this length for all possible allocations of the length to the sub-rules. In this case, the generator will first apply the allocation (4,0,0) to the three rules of the first rule list. Since both the second rule, `BINARY_OPERATOR`, and the third rule, `exp`, cannot generate strings of length 0, this allocation fails. The generator then tries the next possible allocation, (3,0,1). This allocation also fails, so the generator continues incrementing the allocation, until the valid allocation of (2,1,1) is encountered.

The allocation of lengths to nonterminals is recursive, so that the allocation of a portion of the total length to a specific rule causes it to further distribute this length to its constituent rule lists. The end result is a grammar tree for the given length. Figure 3(a) shows a possible grammar tree for the example grammar and a length of 4.

In cases where `yagg` is able to precompute the allowable lengths for a rule, it can rapidly determine whether a given allocation is invalid. For complex recursive grammars, determining valid lengths for a given nonterminal *a priori* is difficult, so the generator also caches information regarding successful allocations for specific rule instances during the generation process.

During the second step, the generator enumerates the possible strings for the current allocation. The entire string is defined by current values of all of the terminals (leaves) in the grammar tree.

In 3(a), the sequence of terminals is `UNARY_OPERATOR NUMBER BINARY_OPERATOR NUMBER`. Terminals are incremented starting with the last terminal. When its strings are exhausted, the next terminal is incremented and the subsequent terminals are reset. This process is similar to numerical counting, where incrementing the number “09” results in an increment in the tens digit and a reset of the ones digit, yielding “10”.

When all strings for a given allocation have been generated, the allocation is incremented and all of the following terminal rules reset. For nonterminals with multiple rule lists, the final increment for one rule list results in the next rule list becoming the active rule list. Therefore, an allocation increment usually results in a new grammar tree, as the current rule list for one or more nonterminals changes.

Figure 3(b) shows the next valid grammar tree after incrementing the allocations for the grammar tree shown in Figure 3(a). In this case, the expansion for the two expressions of the binary operator rule list are different due to the change in allowed lengths for those rules. After their strings have been generated, the generator will replace the binary operator rule list with the next rule list involving the unary operator. The resulting grammar tree is shown in Figure 3(c).

Once a new valid allocation is found, the generation of strings for that grammar tree begins. Generation completes when there are no more possible allocations, and all strings for the final allocation have been generated.

### 4.4 Context-Sensitive Constraints

Context-sensitive constraints arise both in the use of equivalence terminal specifications and in the execution of action blocks. As each string is generated, action blocks are run, possibly invalidating the string as a result of failing some context-sensitive check. In this case, the string increment continues to the next string until a valid one is found, or there are no more strings.

If a valid string was found during the previous call to `Check_For_String()`, the next call will first cause any unaction block to be executed, thereby undoing any state changes. `yagg` also saves and restores copies of the positional parameters to the action block, being careful to handle pointer types appropriately. (The developer must provide copy constructors for any user-defined type.) `yagg` performs this automation so that the programmer need only write unaction blocks to restore any changes to global state.

Equivalence terminals involve context-sensitive analysis of the generated string. One consequence of their semantics is that the enumeration must be performed from the end terminals. For such terminals, their total number of strings depends upon the number of terminals of the same type that appear in the prefix of the string.

## Generation Rate

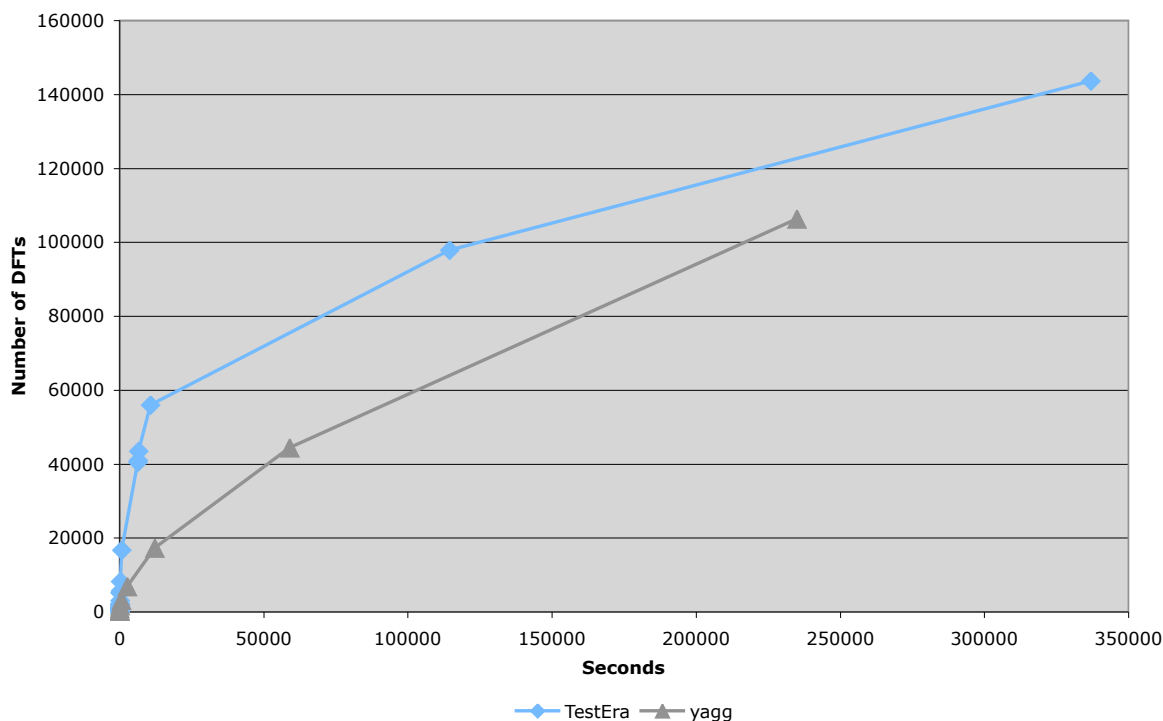


Figure 4: A performance comparison of yagg and TestEra+postprocessor

For example, when generating the third variable for an arithmetic expression whose prefix is “ $x+x+$ ”, it is only necessary to generate two variables, yielding “ $x+x+x$ ” and “ $x+x+y$ ”. On the other hand, a prefix of “ $x+y+$ ” requires three variables to be generated, yielding “ $x+y+x$ ”, “ $x+y+y$ ”, and “ $x+y+z$ ”.

Such context-sensitive constraints complicate the generation of inputs. For nonterminals that have no such action blocks or equivalence terminals, the generated strings will always be the same, so yagg caches the strings as an optimization.

## 5. EVALUATION

In previous work, we evaluated the feasibility of bounded exhaustive testing for nontrivial systems having structurally complex inputs [7, 23]. A substantial part of that research effort was devoted to the development of efficient techniques for input generation. Our collaborators developed a two-phase generation strategy in which a model checker is used to generate the abstract structure of inputs, and a postprocessor combinatorially instantiates the abstract structure to create numerous concrete inputs.

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) [5, 10, 28]. A DFT is a graph in which *basic event* nodes represent component failure events or relevant environmental events, and in which *gate* nodes represent events that occur as functions of other events. For example, an AND gate represents a derived boolean value of its inputs, so that the AND gate is said to be failed if all the inputs have occurred. DFTs are dynamic in that the semantics of some gates, such as spare gates, depends upon the order of failures in the model. DFTs also con-

tain *constraints* that limit the manner in which events occur in the model. For example, the functional dependency (FDEP) constraint indicates that dependent basic events must occur if the trigger event occurs. A sequence enforcing (SEQ) constraint forces failures to occur in a particular order in the model.

DFTs are essentially a graphical language for modeling the reliability of complex systems. Because graphical languages can be cumbersome to work with, reliability engineers have developed an associated textual language for representing DFTs. Like the graphical language, the textual DFT language has a nontrivial syntax that precludes the generation of DFTs using simple context-free grammars. For example, cycles are not allowed in the inputs of gates, spare gate inputs and FDEP dependents can only be basic events, and referenced events must be defined. Other constraints can be found elsewhere [6].

### 5.1 Usability

In our previous work, we used the two-phase generation approach to generate DFTs. The first phase used TestEra, which generated abstract DFTs. These intermediate inputs abstracted away the details of the gate types and basic events, expressing the structure of DFTs in terms of an Alloy specification [14]. 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 we took advantage of the expertise of one of the TestEra developers. This was particularly important for the development of symmetry-breaking predicates that prevent the generation of isomorphic inputs.

The post-processor was developed by hand in Perl. The instan-

tiation of the abstract elements of the abstract DFTs is a straightforward process. While there is some possibility that it can be automated, it is currently a manual process. The two-phase approach has another important role: post-processing can be used to construct concrete inputs that satisfy numerical constraints. Such constraints are typically poorly supported by specification languages in general, and model checkers in particular. Alloy is no exception in this regard.

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 a language like 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.

Unlike YACC, which generates LALR parsers, yagg generates input generators that are fully context-free. One side effect is that ambiguous grammars will have multiple syntax trees, resulting in the same string being generated more than once. Typically programmers address such ambiguity during the development of a parser, when tools such as YACC will warn about shift/reduce or reduce/reduce problems in the grammar.

## 5.2 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 4 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 over time. We hypothesize that the TestEra-based solver is faster due to its symbolic manipulation of the input space. The Alloy model of DFTs is converted internally into a large Boolean expression that is then sent to an off-the-shelf SAT solver. Using heuristics, the solver can manipulate the model to more efficiently find instances of the model. 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 ap-

proach. (It should be noted that the underlying SAT solver, mChaff [20], 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.

## 6. 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. We believe that this tool can help developers who wish to employ exhaustive testing on their software.

One could argue that declarative languages like Alloy, while unfamiliar to most programmers, are easier to use due to their terseness. The design goals of the yagg tool have instead emphasized compatibility with potentially pre-existing constraint checks expressed in an imperative language. Our underlying assumption, which needs to be tested, is that reuse or writing of a larger amount of imperative code is easier than learning and using a more terse declarative language.

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 data 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.

## 7. REFERENCES

- [1] Godmar Back. DataScript—a specification and scripting language for binary data. In *Proceedings of the ACM Conference on Generative Programming and Component Engineering Proceedings*, pages 66–77, Pittsburgh, PA, 6–8 October 2002. Springer.
- [2] 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.
- [3] Franco Bazzichi and Ippolito Spadafora. An automatic generator for compiler testing. *IEEE Transactions on Software Engineering*, 8(4):343–53, July 1982.

- [4] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, Rome, Italy, 22–24 July 2002. IEEE.
- [5] Mark A. Boyd. *Dynamic Fault Tree Models: Techniques for Analysis of Advanced Fault Tolerant Computer Systems*. PhD thesis, Duke University, Department of Computer Science, April 1991.
- [6] David Coppit. *Engineering Modeling and Analysis: Sound Methods and Effective Tools*. PhD thesis, The University of Virginia, Charlottesville, Virginia, January 2003. URL: <http://www.cs.wm.edu/~coppit/papers/dissertation.pdf>.
- [7] 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. To appear.
- [8] François Désarménien. The Parse::Yapp homepage. URL: <http://search.cpan.org/~fdesar/Parse-Yapp/>.
- [9] Mark-Jason Dominus. The Text::Template homepage. URL: <http://www.plover.com/~mjd/perl/Template/>.
- [10] Joanne Bechta Dugan, Salvatore Bavuso, and Mark Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–77, September 1992.
- [11] 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.
- [12] K. V. Hanford. Automatic generation of test cases. *IBM System Journal*, 9(4):242–57, 1970.
- [13] Robert M. Hierons. Testing from a Z specification. *Software Testing, Verification, and Reliability*, 7:19–33, 1997.
- [14] Daniel Jackson. Micromodels of software: Modelling and analysis with Alloy. URL: <http://sdg.lcs.mit.edu/alloy/reference-manual.pdf>.
- [15] Daniel Jackson, I. Schechter, 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.
- [16] S. C. Johnson. YACC — Yet another compiler-compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, N.J., 1975.
- [17] M. E. Lesk and E. Schmidt. Lex — A lexical analyzer generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, N.J., 1975.
- [18] 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.
- [19] Peter M. Maurer. The design and implementation of a grammar-based data generator. *Software Practice and Experience*, 22(3):223–44, March 1992.
- [20] 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.
- [21] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the 2nd International Conference on the Unified Modeling Language*, pages 416–29, Fort Collins, CO, 28–30 October 1999. Springer.
- [22] P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3):366–75, 1972.
- [23] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 133–42, Boston, Massachusetts, 11–14 July 2004. IEEE.
- [24] The Free Software Foundation. The Bison homepage. URL: <http://www.gnu.org/software/bison/>.
- [25] The Free Software Foundation. The Flex homepage. URL: <http://www.gnu.org/software/flex/>.
- [26] The Free Software Foundation. The GNU Make homepage. URL: <http://www.gnu.org/software/make/>.
- [27] The rsync team. The rsync homepage. URL: <http://rsync.samba.org/>.
- [28] W. E. Vesely, Joanne Dugan, Joseph Fragola, Joseph Minarick III, and Jan Railsback. *Fault Tree Handbook with Aerospace Applications*. National Aeronautics and Space Administration, August 2002.
- [29] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O’Reilly & Associates, Inc., Newton, MA, July 2000.