Directed Test Generation using Symbolic Grammars*

Rupak Majumdar Ru-Gang Xu
CS Department, UC Los Angeles
rupak@cs.ucla.edu, rxu@cs.ucla.edu

ABSTRACT

We present CESE, a tool that combines exhaustive enumeration of test inputs from a structured domain with symbolic execution driven test generation. We target programs whose valid inputs are determined by some context free grammar. We abstract the concrete input syntax with *symbolic grammars*, where some original tokens are replaced with symbolic constants. This reduces the set of input strings that must be enumerated exhaustively. For each enumerated input string, which may contain symbolic constants, symbolic execution based test generation instantiates the constants based on program execution paths. The "template" generated by enumerating valid strings reduces the burden on the symbolic execution to generate syntactically valid inputs and helps exercise interesting code paths. Together, symbolic grammars provide a link between exhaustive enumeration of valid inputs and execution-directed symbolic test generation.

Preliminary experiments with CESE show that the combination achieves better coverage than both pure enumerative test generation and pure directed symbolic test generation, in orders of magnitude less time and number of generated inputs. In addition, CESE is able to automatically generate inputs that achieve coverage within 10% of manually constructed tests.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Reliability, Verification

Keywords: symbolic grammars, grammar based testing, random testing, concolic execution, testing C programs

1. INTRODUCTION

We consider the problem of automatic and comprehensive test input generation for large software programs where valid inputs to the system come from some structured domain. Examples of such software systems are compilers or command processors, which accept inputs that form valid strings in some context free language, or business applications which process input described by some XML schema. There are two predominant ways to automatically generate test inputs for such systems: enumerative and symbolic. In *enumerative* test generation, all inputs satisfying a certain input specification (*e.g.,* , grammar for a parser or an XML schema) are enumerated (up to some bounded size), and the program is executed on all the inputs [9, 22]. In *symbolic* test generation [8, 21, 36, 3, 16], the program is executed on symbolic rather than (or in addition to [16, 32]) concrete inputs, and a set of constraints on the symbolic inputs is collected along an execution trace. A constraint solver is then used to generate test inputs that satisfy the symbolic constraints. The resulting test inputs are guaranteed to force the program execution along the path chosen by the symbolic execution.

Specification-based exhaustive enumeration is guaranteed to provide *valid inputs* to the program. This ensures that the application goes beyond the parsing and input sanitizing phase and executes along deeper paths. Unfortunately, exhaustive enumeration does not distinguish between different observable behaviors produced by the inputs. Thus, a large set of redundant tests may be generated, each of which has exactly the same execution behavior on the program. Also, for almost all nontrivial programs, the set of possible valid inputs is too large to completely enumerate, and in practice, one explores a random sampling of the input space, through some form of random or biased test input generation. This is not comprehensive: very often, the probability that random testing exercises program corner cases, where many bugs lurk, is astronomically small. In summary, specification-based exhaustive enumeration, while *selective*, in that it generates test inputs from the program's expected input domain, is *not directed*, in that the actual execution paths are not considered in the test generation.

In contrast, test generation based on symbolic execution is *directed*, exploiting path equivalences, and systematically exploring new paths. However, most symbolic execution implementations are *not selective*: they start with an unstructured buffer of symbolic variables, and hope to extract the structure of the input by looking at the tests executed along the path. While theoretically complete, symbolic techniques are expensive, and ultimately limited by the capacity of the symbolic engine. In practice, a symbolic test generator for a compiler stays "forever" within the many paths of the parser, generating incorrect inputs one after another, but exploring only novel parse error paths!

*This research is sponsored in part by the NSF grants CCF-0427202, CCF-0546170, CCF-0702743, and CNS-0720881.

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

We present a test input generation algorithm that combines the advantages of selective enumerative test generation and directed symbolic test generation. There is a tension between the two techniques. For any single input, program execution is orders of magnitude faster than symbolic exploration, hence one should push as much work to enumerative testing as possible. On the other hand, the number of possible inputs to be enumerated is astronomical, so one should push as much work as possible to the symbolic engine to explore only non-redundant computations. Our solution is the use of *symbolic grammars* that balance the two competing requirements. Our test generation algorithm (1) transforms a grammar specifying input format into a symbolic grammar, (2) enumerates the set of valid strings in the symbolic grammar using enumerative techniques, and (3) runs symbolic test generation on the symbolic strings enumerated.

Take a grammar G for arithmetic expressions (numbers, or the sum or difference of two arithmetic expressions):

$$\begin{array}{lcl} \texttt{exp} & ::= & \texttt{num} \mid \texttt{exp} + \texttt{exp} \mid \texttt{exp} - \texttt{exp} \\ \texttt{num} & ::= & [0-9] \end{array}$$

For an input of size 3, enumerative techniques will generate all 210 valid strings of the form 0, ..., 9 (for integer constants), and $0+0$, $0-0$, $0+1$, $0-1$, ..., $9+9$, $9-9$. Symbolic techniques will start with three symbolic variables, and generate a large set of invalid inputs (e.g., "+00", "+–") to explore (the large number of) character-by-character comparisons in the lexer and error paths in the parser. In contrast, a *symbolic grammar* G' for G can replace the production of `num` with

$$\texttt{num} ::= \alpha$$

where α is a symbolic constant whose value is instantiated during symbolic exploration based on comparisons in the code. With this transformation, the number of possible valid strings of length 3 are α, $\alpha+\alpha$, and $\alpha-\alpha$. At this point, we can run symbolic execution on the three inputs where symbolic constants are instantiated with respect to unique program paths.

Symbolic grammars enable several orders-of-magnitude decrease in the number of strings to be enumerated, and for each enumerated string, the symbolic constants generate enough non-determinism for the symbolic test generation to explore all paths of the program. Consequently, the use of symbolic grammars lets us profitably combine enumerative and symbolic test generation techniques to get a combined test generation algorithm whose performance should be much better than either alone. This last claim must be empirically validated. We have implemented CESE (Concolic Execution with Selective Enumeration), a tool that implements test generation using symbolic grammars for C programs that specify their input syntax using lex and yacc, on top of the Yagg string generator [9] and Cute concolic execution [32] tools.

We have applied our implementation to generate test inputs for a set of open source programs. In our initial experiments on a calculator for arithmetic expressions (used as an example application in many yacc tutorials), CESE outperformed both strictly enumerative and strictly symbolic test generation. The symbolic grammar had two orders of magnitude fewer strings to be enumerated. With symbolic grammar-based enumeration, CESE explored two orders of magnitude fewer inputs than Cute for input buffers of size four, and could finish enumeration for larger buffers when Cute could not finish within 5 hours. Similar trends were borne out in other experiments. Overall, CESE was able to achieve an average 10% more branch coverage than Cute in a 30 minute testing budget. Further, limit experiments where Cute was run for 5 hours showed that the branch coverage obtained by Cute saturated (*i.e.,* , did not significantly improve over the coverage obtained in 30 minutes), and remained approximately 9% less than CESE running for 30 minutes. Further, for the programs in our suite that came with manual testcases, we saw that branch coverage obtained by CESE was within 10% of coverage with manual tests. This difference could be attributed to program behaviors that only manifest with larger input buffers. We find this impressive: in spite of enumerating very small input buffers, CESE was able to come within the same ballpark as carefully crafted manual tests. In comparison to pure enumerative (grammar-based) input generation, CESE generated several orders of magnitude fewer inputs, and achieved slightly better (6% better) coverage under the same testing budget. Since generated tests are often added to regression suites, the many fewer tests generated by CESE (and consequently, the much lower test execution time) indicates a win for CESE. We also used CESE to check for buffer overflows, in particular, to check if a known buffer overflow in the path resolution function of the `wuftpd` FTP server can be detected. CESE found the bug in four minutes, whereas Cute timed out without finding the bug in 13 hours. The specific configuration that leads to this bug requires a buffer of over 1000 bytes, making it outside the scope of exhaustive enumeration, and making the odds against random testing astronomically high. These initial results are clearly indicative that CESE is a scalable and useful technique for automated comprehensive test generation, and can match or outperform several known test input generation algorithms.

2. EXAMPLE

We introduce and motivate our technique by testing a calculator example *SimpleCalc* that is seen in many tutorials for yacc [19] and lex [23]. The *SimpleCalc* implementation consists of 1826 lines of generated C code. The grammar for *SimpleCalc* inputs is shown below.

$$\begin{array}{lcl} \text{Expressions } e & ::= & (e) \mid e * e \mid e/e \mid e \,\%\, e \mid e + e \mid e - e \\ & & \mid e \vee e \mid e \wedge e \mid -e \mid l \mid n \\ \text{Letters } l & ::= & [a-zA-Z] \\ \text{Numbers } n & ::= & [0-9] \end{array}$$

The program takes an arithmetic expression with letters as variables, various numerical operators, parentheses for precedence, and logical operators. The calculator implementation replaces letters with numbers that have been recorded in an array. Numerical and logical operators are directly applied, and precedence is handled during parsing. This implementation contains bugs: the *SimpleCalc* implementation forgets to check for division or modulus by zero.

We test *SimpleCalc* with a fixed input buffer of four bytes called *input*. We compare and contrast random testing, test generation using concolic execution using the tool Cute, and concolic testing with selective enumeration using CESE. We restrict the size of our buffer to four so we can exhaustively test all program paths using both Cute and CESE. Although it is generally infeasible to run either Cute or CESE to completeness for large inputs or large programs, this small example clearly highlights the differences between naive con-

colic execution, concolic execution with selective enumeration, random testing, and specification-guided testing using concrete grammars. We compare the branch coverage obtained for both Cute and CESE, where *branch coverage* is the percentage of branches executed, and whether the bugs can be found. We also examine the effect of increasing the input buffer size on these techniques.

2.1 Random Testing

With an input size of four bytes, there are $(2^8)^4 = 2^{32}$ unique inputs. The input space is too large for exhaustive testing all inputs. An automatic way of tackling this problem is to randomly choose inputs. However, we claim that random testing is not effective for this examples because the chances of hitting bugs are very low.

Based on the *SimpleCalc* grammar, there are 80,910 valid strings of size four, 27,032 of size three, 62 of size two and 62 of size one. To calculate the number of valid input buffers, we take account of the string terminator. With an input of size one, the string terminator must be at *input*[1]. The contents at *input*[2] and *input*[3] do not matter. Therefore, there is a total of $2^8 \cdot 2^8 \cdot 62 = 4{,}187{,}046$ inputs representing valid strings of size one. Following the same calculation, we have 15,872 inputs representing valid strings of size two, 27,032 of size three and 80,910 of size four, totaling close to 4.2 million valid inputs. Therefore, every input has around a 0.1% chance of being a syntactically correct input and the majority of these inputs will be only of size one, thus unlikely to exercise any interesting paths.

Generating an input that will show buggy behavior in this calculator is smaller. This implementation does not check for divide by zero errors, therefore operations dividing by zero or modulo by zero result in runtime exceptions. Valid strings containing "/0" or "%0" result in this error. For valid inputs of size four or less, there are only 372 inputs that demonstrate the error. Thus, random testing has a 0.000009% chance of hitting bugs. In fact, even if *SimpleCalc* is tested with 8 million random inputs, there is only a 50% chance that a bug causing string would have been generated. The problem, as is well-known, is that random testing is neither selective nor directed.

2.2 Constrained Exhaustive Enumeration

Specification-based test generation improves the pitfalls of random testing by generating inputs that are guaranteed to satisfy certain well-formedness specifications [9, 17, 5, 20]. In particular, there are test input generators that take as input a grammar (written, *e.g.,* in yacc) describing valid inputs, and generates test cases that satisfy the grammar [9, 25, 22]. Usually, these techniques exhaustively enumerate all inputs satisfying the specification, and test the program on all such inputs. Unfortunately, even for simple input specifications such as our grammar for *SimpleCalc*, the space of valid inputs is very big. As Table 1 demonstrates, for the *SimpleCalc* example, the number of valid strings for an input buffer of size six is already $187,765,078$. Enumerating and testing this large space of inputs is expensive. Moreover, certain errors may only be exhibited when the input buffer is much larger. Exhaustive enumeration can generate many equivalent test cases, *i.e.,* tests that have the same observable behavior on the program. In this example, the grammar-based input generator Yagg [9] generates the tests $0 + 0$, $0 + 1$, $0 + 2$, etc., all of which exercise the identical program path. Thus, while the enumerative strategy for specification-based testing is *selective*, it is not *directed*.

2.3 Symbolic or Concolic Execution

An alternative is symbolic execution of the code [21, 8], where the program is executed on symbolic inputs, and satisfying assignments to constraints collected along a program path comprise new test inputs. Recent attempts combine symbolic execution with concrete random execution of the code [16, 32, 6] (called "concolic execution" in [32]). The concrete execution allows the symbolic execution to simplify constraints based on the concrete values along the run. Symbolic execution based test generation is *directed*: test inputs are generated by systematically exploring program paths at the symbolic level, and these inputs are then guaranteed to execute along pre-determined paths. Thus, the set of test inputs generated are not redundant: each leads to a different program path.

Unfortunately, current implementations of concolic execution based test generation are not selective: test inputs are generated randomly, and iteratively refined using symbolic constraints. While theoretically complete in the limit, in practice, the lack of selectivity is a serious problem, and a very large number of inputs must be generated to reach the part of the code not related to input error handling. This leads to poor coverage for most realistic testing budgets.

We test the capability of symbolic execution based test generation on the calculator example, using Cute, an implementation of concolic execution [32]. To test *SimpleCalc* with Cute, we created a symbolic input buffer of size four. Cute then exhaustively generates all paths in the program by iteratively finding satisfying assignments to constraints that lead to paths that have not yet been covered. Unfortunately, the code for parsing examines all possible values for its input characters. For lex, this operation is represented by a table lookup. Figure 1 shows branches that are equivalent to this table lookup. From just these 10 branches, Cute can derive 10^4 unique paths for a size four buffer. Coupled with the other branches in the code, Cute needed to explore a total of 248,523 inputs, taking 30 minutes. This worsens as the input size increases. As Table 1 shows, we could not finish exhaustive testing of program paths for buffers greater than four characters even after 5 hours.

2.4 This Paper: CESE

The main idea of CESE is to combine the selectiveness of specification-guided test generation with the directedness of symbolic or concolic test generation. To do this, we introduce *symbolic grammars*. It will be convenient for us to consider context free grammars where the terminal symbols are regular expressions rather than individual characters from an alphabet. A symbolic grammar for a (concrete) grammar replaces some terminals of the grammar with a symbolic constant. Each string in the symbolic grammar represents a set of strings, where each symbolic constant is substituted with a string in the regular expression which it represents.

For example, a symbolic grammar G'_{calc} for *SimpleCalc* replaces the concrete productions for letters and numbers with symbolic placeholders:

$$\begin{array}{lcl} \text{Letters } l & ::= & \alpha \\ \text{Numbers } n & ::= & \beta \end{array}$$

	Number of Inputs by Technique					Time		Coverage
Length	**Grammar**	**Exhaustive**	Cute	**Sym Grammar**	CESE	Cute	CESE	
1	62	2^8	21	1	21	0.5s	4s	124/344 = 36%
2	124	2^{16}	247	2	40	2s	3s	179/344 = 52%
3	27,156	2^{24}	2,515	11	1,711	20s	24s	186/344 = 54%
4	108,066	2^{32}	248,532	35	6,611	30m	3m	194/344 = 56%
5	47,008,834	2^{40}	n/a	201	260,792	n/a	1h	200/344 = 58%
6	187,765,078	2^{48}	n/a	652	1,492,802	n/a	3h	200/344 = 58%

Table 1: Effect of input size. Length is the maximum size of the input buffer. In the Number of Inputs by Technique column: Grammar denotes the number of syntactically valid strings, Exhaustive denotes the number of unique buffers, Cute gives the number of inputs generated by Cute, Sym Grammar gives the number of strings in the symbolic grammar, CESE gives the number of inputs generated by CESE. Time gives the execution time for CESE denoted by CESE and the execution time for Cute denoted by Cute in seconds (s), minutes (m) or hours (h). Coverage shows the branch coverage for both Cute and CESE in the first 4 rows, but just for CESE in the last two. Cute did not terminate within 5 hours for those tests so such entries are marked as n/a.

where α and β are symbolic constants. With this change, the number of strings of a certain length that can be generated by the grammar reduces significantly. For example, instead of the 100 different strings "0/0", "0/1", ..., "9/9" representing division, we now have just one symbolic string "β_1/β_2" representing all these concrete strings. Note that we use subscripts for the different occurrences of the symbolic variables, each occurrence of a symbolic constant is instantiated separately.

The original program does not know about symbolic constants so our test generation algorithm must instantiate symbolic constants with actual constants. This instantiation can be performed in a *directed* way by treating symbolic constants as unconstrained symbolic values to be filled in by concolic execution. Think of a string generated by a symbolic grammar as a string with "holes" for certain terminals. These holes are filled in by a concolic execution, depending on branches executed within the code. Together, the reduction in the number of possible strings in the language enables exhaustive enumeration to scale —thus providing selectivity— and concolic execution with the symbolic constants enables exploration of non-redundant strings — thus providing directedness.

This is the basic idea of CESE. We convert the concrete grammar to a symbolic grammar by replacing certain lexical tokens with symbolic constants. What tokens to replace is decided by a simple heuristic. If the token represents one concrete string (*e.g.,* lexical tokens corresponding to program keywords or operators), it is not replaced. On the other hand, if the lexical token corresponds to an unbounded set of concrete strings (*e.g.,* variable names, numbers), we replace it with a symbolic constant. Second, we exhaustively enumerate all symbolic strings from the symbolic grammar (G'_{calc} in the example) up to a certain size. Third, for each (symbolic) string, we use concolic execution to perform directed testing, where each symbolic constant is considered to be an unconstrained input to be solved for.

For example, "$\alpha_1 + \alpha_2$" is run by forcing only the second byte to be '+' and allowing concolic execution to generate values for α_1 and α_2 that exercise different paths. For this particular symbolic input, concolic execution exercised 188 unique paths. By working on the symbolic grammar, CESE has replaced 3,844 possible runs (corresponding to the valid

```
if (*yy_cp >= 0 && *yy_cp <= 0) yy_c = 0;
if (*yy_cp >= 1 && *yy_cp <= 7) yy_c = 1;
if (*yy_cp >= 8 && *yy_cp <= 8) yy_c = 2;
if (*yy_cp >= 9 && *yy_cp <= 31) yy_c = 1;
if (*yy_cp >= 32 && *yy_cp <= 32) yy_c = 3;
if (*yy_cp >= 33 && *yy_cp <= 47) yy_c = 1;
if (*yy_cp >= 48 && *yy_cp <= 57) yy_c = 4;
if (*yy_cp >= 58 && *yy_cp <= 96) yy_c = 1;
if (*yy_cp >= 97 && *yy_cp <= 122) yy_c = 5;
if (*yy_cp >= 123 && *yy_cp <= 255) yy_c = 1;
```

Figure 1: Calculator Lexer.

grammar strings) with 188 concolic executions. Compared to Cute, CESE gets the same coverage for substantially fewer inputs (6,611 versus 248,532) and an order of magnitude less time (3 minutes, versus 30 minutes). Table 1 shows the number of symbolic strings generated and also the number of concrete inputs generated from all those symbolic strings.

While the number of inputs still grows as the input buffer size increases, the significant reduction in the input space by moving to a symbolic grammar allows us to exhaustively search larger inputs. In further experiments detailed in Section 4, we have found that this combination of selective symbolic test input generation together with directed search is essential in scaling concolic test generation to real examples.

3. THE CESE APPROACH

3.1 Programs

We describe our algorithm on an idealized imperative language. The operations of the programming language consist of labeled statements $\ell : s$. Labels correspond to instruction addresses. A statement is either (1) the normal termination statement halt or the abnormal program termination statement abort, (2) an *input statement* $\ell : l := \mathsf{input}(k, G)$ that copies an external character buffer of size k into a buffer l of size at least k, where the input is expected (but not guaranteed) to be from a context free language defined by the context free grammar G, (3) an assignment $l := e$ where l is an lvalue and e is a side-effect free expression, (4) a conditional statement $\mathsf{if}(e)\mathsf{goto}\ \ell$ where e is a side-effect free expression and ℓ is a program label. Execution begins at the program

label ℓ_0. For a labeled assignment statement $\ell : l := e$, or input statement $\ell : l := \mathsf{input}(k, G)$ we assume $\ell + 1$ is a valid label, and for a labeled conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$ we assume both ℓ' and $\ell + 1$ are valid program labels.

The set of *data values* consists of program memory addresses and character values. The semantics of the program is given using a *memory* consisting of a mapping from program addresses to values. Execution starts from the initial memory M_0 which maps all addresses to some default value in their domain. Given a memory M, we write $M[m \mapsto v]$ for the memory that maps the address m to the value v and maps all other addresses m' to $M(m')$.

For an assignment statement $\ell : l := e$, the address m of the left-hand side l, where the result is to be stored, and the expression e is evaluated to a concrete value v in the context of the current memory M, the memory is updated to $M[m \mapsto v]$, and the new program location is $\ell+1$. For an input statement $\ell : l := \mathsf{input}(k, G)$, the transition relation updates the memory M to the memory $M[m \mapsto v]$ where m the base address of the buffer l, and v is a nondeterministically chosen character buffer of size k, and the new location is $\ell + 1$. For a conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$, the expression e is evaluated in the current memory M, and if the evaluated value is zero, the new program location is ℓ' while if the value is non-zero, the new location is $\ell+1$. In either case, the new memory is identical to the old one. Execution terminates normally if the current statement is halt, abnormally if the current statement is abort.

3.2 Concolic Test Generation

We briefly recapitulate the concolic testing algorithm from [16, 32, 6]. Concolic testing performs symbolic execution of the program together with its concrete execution. It maintains a *symbolic memory map* μ and a *symbolic constraint* ξ in addition to the (concrete) memory. These are filled in during the course of execution. The symbolic memory map is a mapping from concrete memory addresses to symbolic expressions, and the symbolic constraint is a first order formula over symbolic terms. The details of the construction of the symbolic memory and constraints is standard [36, 16, 32]. That is, at every statement $\ell : l := \mathsf{input}(k, G)$, the symbolic memory map μ introduces a mapping $m \mapsto \langle \alpha_1, \ldots, \alpha_k \rangle$ from the address m of l to a tuple of k fresh symbolic values $\alpha_1, \ldots, \alpha_k$, and at every assignment $\ell : l := e$, the symbolic memory map updates the mapping of the address m of l to $\mu(e)$, the symbolic expression obtained by evaluating e in the current symbolic memory. As an optimization, a mapping is maintained in the symbolic memory for an address m iff the value at m is a symbolic expression, if it is a concrete value, the mapping is not maintained in μ. The concrete values of the variables (available from the memory map M) are used to simplify $\mu(e)$ by substituting concrete values for symbolic ones whenever the symbolic expressions grow too large or go beyond the theory that can be handled by the symbolic decision procedures.

The symbolic constraint ξ is initially `true`. At every conditional statement $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$, if the execution takes the then branch, the symbolic constraint ξ is updated to $\xi \wedge (\mu(e) \neq 0)$ and if the execution takes the else branch, the symbolic constraint ξ is updated to $\xi \wedge (\mu(e) = 0)$. Thus, ξ denotes a logical formula over the symbolic input values that the concrete inputs are required to satisfy to execute the path executed so far.

Given a concolic program execution, concolic testing generates a new test in the following way. It selects a conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$ along the path that was executed. Let ξ_ℓ be the symbolic constraint just before executing this instruction and ξ_e be the constraint generated by the execution of this instruction. Using a decision procedure, concolic testing finds a satisfying assignment for the constraint $\xi_\ell \wedge \neg\xi_e$. The property of a satisfying assignment is that if these inputs are provided at each input statement, then the new execution will follow the old execution up to the location ℓ, but then take the conditional branch opposite to the one taken by the old execution.

3.3 Symbolic Grammars

Let Σ be a finite alphabet. A *terminal* is a regular expression over Σ. We define a *grammar* $G = (V_t, V_n, R, S)$ where V_t is a finite set of terminals, V_n is a finite set of *variables*, $R \subseteq V_n \times (V_n \cup V_t)^*$ is a finite set of production rules, and $S \in V_n$ is a distinguished start variable. The language $L(G) \subseteq \Sigma^*$ of the grammar G is defined in the usual way [33]. The language $L_h(G) \subseteq \Sigma^*$ of the grammar $G = (V_t, V_n, R, S)$ is defined as all strings derived from h applications of any of the productions rules R from the start variable S. A word $w \in L_h(G)$ has a *height* of h.

Let $\alpha_1, \ldots, \alpha_k$ be k symbolic names not in Σ. We assume each α_i stands for the regular language $\{\alpha_i\}$. A *symbolic grammar* G' for a grammar G w.r.t. terminals $T = \{t_1, \ldots, t_k\} \subseteq V_t$ is the grammar $(V_t \setminus T \cup \{\alpha_1, \ldots, \alpha_k\}, V_n, R[\alpha_i/t_i], S)$ where $R[\alpha_i/t_i]$ substitutes α_i for each occurrence of t_i for $i \in \{1, \ldots, k\}$. The language of the symbolic grammar G' is a subset of $(\Sigma \cup \{\alpha_1, \ldots, \alpha_k\})^*$. Notice that a string can now contain symbolic constants.

A symbolic grammar G' *abstracts* a concrete grammar G in the following sense. For any string $w \in L(G)$, there exists $w'(\beta_1, \ldots, \beta_k) \in L(G')$ with symbolic constants β_1, ..., β_k replacing terminals t_1, ..., t_k such that there exist strings $a_1 \in L(t_1)$, ..., $a_k \in L(t_k)$ such that $w = w'[\beta_1/a_1, \ldots, \beta_k/a_k]$.

Given a (concrete or symbolic) grammar G and a height h, all possible strings in $L_h(G)$ can be enumerated by dynamic programming [9].

3.4 The CESE Algorithm

The CESE algorithm has four phases: symbolic grammar construction, exhaustive enumeration, program instrumentation, and concolic execution. Let P be a program with input statements $m_i := \mathsf{input}(k_i, G_i)$ for some range of indices i.

For the first phase, we construct a symbolic grammar G'_i for each concrete grammar G_i. The symbolic grammar construction uses the following heuristic. If a lexical token is a constant string (equivalently, if the regular expression defines a singleton language), then no symbolic constants are generated. Otherwise, we distinguish between finite regular languages and infinite regular languages. This distinction can be checked by looking for cycles in the derived automaton. For a finite regular language with bound k on the length of strings, we introduce k symbolic variables $\alpha_1, \ldots, \alpha_k$ and replace the token with the k sequences α_1, $\alpha_1\alpha_2$, ..., $\alpha_1 \ldots \alpha_k$. For an infinite regular language, we replace the token with the symbolic regular language α^* denoting any number of symbolic constants.

In our experiments, the tokens either defined regular languages with one letter strings (*e.g.,* tokens for single-letter variable names in *SimpleCalc*), or infinite regular languages (*e.g.,* tokens for numbers).

However any subroutine that converts a concrete grammar to an abstracting symbolic grammar can be used. There is a trade-off between the number of symbolic strings with the number of symbolic variables in each string. As the number of symbolic variables increases, the number of valid symbolic strings decreases. For example, the coarsest abstraction is an unbounded number of symbolic letters. This symbolic grammar only contains four strings for an input of size four: α_1, $\alpha_1\alpha_2$, $\alpha_1\alpha_2\alpha_3$ and $\alpha_1\alpha_2\alpha_3\alpha_4$. However using this abstraction is equivalent to just using concolic execution.

Once a symbolic grammar is constructed, we use exhaustive enumeration techniques [9, 22, 25] to generate strings from the grammar G'_i up to height h_i. The generated strings have both constant symbols and symbolic constants. For each choice $w \in L_{h_i}(G'_i)$ and the length of w is less than k_i, we replace the statement $m_i = \mathsf{input}(k_i, G_i)$ with the loop:[1]

$$\textbf{for } j = 0 \textbf{ to } k_i - 1 \textbf{ do } m_i[j] \ := \ \gamma[j]$$

where $\gamma[j] = w[j]$ if $w[j]$ is a constant symbol, and $\gamma[j] = \mathsf{input}(1, \cdot)$ if $w[j]$ is a symbolic constant. Here, $\mathsf{input}(1, \cdot)$ generates a single character (and we ignore the grammar component). The effect of the loop is to only retain the symbolic constants in the string as inputs, while instantiating all constant symbols.

Finally, we perform concolic execution on this instrumented program.

The correctness of the CESE algorithm is defined relative to the Cute algorithm and the algorithm that enumerates all valid strings and executes the program on each string. Specifically, for any program P (that generates exclusively constraints within the capability of the underlying constraint solver), the set of paths explored by Cute on *valid* inputs (an input $\mathsf{input}(k, G)$ is valid if the k characters do form a string in $L(G)$) is exactly the same as the set of paths explored by CESE. Further, this set is exactly the set of paths explored by exhaustive enumeration of all strings from G and executing the program on each string.

4. EXPERIMENTS

CESE was implemented for programs that use yacc and lex to describe their inputs. Both symbolic grammar generation and symbolic string generation were automatic. We used Yagg [9] to automatically generate symbolic strings from our symbolic yacc and lex grammars and Cute [32] as our concolic testing engine. Cute was modified to handle reasoning about statically allocated arrays by replacing those array accesses by branches, as seen in Figure 1. We used `lp_solve` [1] as our underlying linear constraint solver. For real applications, there are rarely resources to explore all symbolic strings, therefore, we can choose which symbolic inputs to use. In the implementation, we sorted the inputs by the number of symbolic constants in the input. This optimization on the average increases coverage by 3% in our 30 minute experiments.

[1]Our basic imperative language does not have a **for** loop. However, we write this for loop for readability. This can easily be converted to more basic control flow in our language.

We ran two sets of experiments to test coverage and bug finding. Section 4.1 compares the effectiveness of CESE, naive concolic testing, random testing and specification based testing in branch coverage. Section 4.2 describes how concolic testing and CESE can find a deep buffer overflow bug. All experiments were performed on a MacBook Pro 2.33 Ghz Intel Core 2 Duo with 2GB RAM running Mac OS X 10.4.8.

4.1 Coverage

Our first set of experiments measured branch coverage. We can distinguish a branch statically (*i.e.,* location in the code) or dynamically (*i.e.,* location on an executed path). Branch coverage is statically unique branches executed over all runs divide by the total number of branches in the program. For all experiments, CESE distinguished each branch dynamically to explore program paths, but then measured the number of statically unique branches covered. CESE can also explore paths based on distinguishing static branches only but the measured branch coverage is usually significantly reduced in that case[15].

We tested five programs `bc`, `lua`, `logictree`, `cuetools`, and `wuftpd`. `bc` is the popular UNIX calculator. `lua` is an interpreter. `logictree` is a logical formula solver. `cuetools` is an API for playlists. `wuftpd` is a popular FTP server. These programs were modified to take a buffer as an input and were linked with a concolic execution aware string library.

Each program has several command line and configuration options. We restrict the programs to have their default configurations and do not explore the alternate configurations. We ran CESE on each program for 30 minutes. For each program, CESE generated words with up to h applications of the production rules of the symbolic grammar, where the parameter h was chosen so that all words that can be derived with $h-1$ applications, would be explored within 30 minutes. Table 5 shows the values of h in the **Height** column. Also note that the generated inputs can be used independently of CESE as part of a regression suite. All inputs generated by CESE for all experiments can be rerun without the symbolic execution and constraint solving in less than 5 minutes. We focus on measuring test generation for the default configuration so we may compare against other approaches.

Manual Testing. Each program has various command line or configuration options, but for all experiments only the default configuration was used. Since we do not exercise all configuration options, full branch coverage is not possible. To estimate the maximum possible branch coverage based on the default configuration, we measure the branch coverage of test cases created by the program developers. We found testcases for all programs except for `wuftpd`. All manual testcases were relatively complex and required substantial knowledge of the program to create. These testcases included mathematical algorithms, sorting algorithms, and a large CD playlist.

Table 2 shows how CESE running for 30 minutes compares to manually created test cases. CESE's automatically generated inputs have 10% less total coverage than the manual tests. We found this quite remarkable considering that two of these programs were language interpreters that included large sets of library functions that were unspecified in the grammar. Also, to our surprise, CESE performed slightly

better in the `cuetools` testcase, because the developer only considered certain types of music lists and did not utilize the complete grammar. In the other programs, CESE was not as effective as manually created tests because CESE did not use a large enough input buffer or did not have time to enumeratively explore all symbolic constants.

Naive Concolic Testing. We used Cute [32] for the comparison. For all our Cute experiments, we chose the input size to provide the best coverage for each 30 minute run. Input size was 10 for all Cute experiments. Table 3 shows the results. On average, CESE had a 10% improvement over Cute. Usually, CESE generates fewer inputs in the allotted time, because CESE explores deeper paths resulting in longer execution times while runs generated by Cute are short runs resulting from parse errors. However in `wuftpd`, Cute generated less inputs, because the number of symbolic constants and their constraints overwhelmed the constraint solver, causing it to timeout. As seen in Table 5, CESE could generate longer input strings with significantly less symbolic variables. Cute required all of the input to be symbolic therefore creating a burden in the constraint solving and limiting the input size.

We also investigated how long it would take naive concolic test generation to achieve the same amount of coverage as running CESE for 30 minutes. Cute cannot get close to the same coverage as CESE even when given ten times as much time. We ran Cute for five hours on each program. There was only a slight increase (1%) in average coverage – still 9% less coverage than running CESE for 30 minutes. Note that for programs requiring larger valid strings such as cuetools, there was no improvement. Cute is stuck in the parsing code – doing a search that is exponential in size of the input. CESE on the other hand, essentially skips a large part of the parsing code and its performance instead depends on the number of production rules and the number of symbolic variables in the symbolic grammar.

Specification-Based Testing. We used the concrete grammar to exhaustively generate inputs with up to h applications of the production rules where h was chosen such that we could explore all inputs with $h-1$ applications. Table 5 shows h per program in the **Height** column. Table 4 shows the coverage for 30 minute runs of grammar based testing. Grammar based testing generates more inputs than CESE in the same amount of time, but CESE explores inputs with more height, therefore, more complex and longer paths. The introduction of symbolic letters in CESE reduces the input space without sacrificing coverage. Normally, this reduction is so significant that the cost of symbolic execution is worth it (*i.e.,* for `bc`, there were 790 CESE inputs of height 3 but 257074 concrete words of height 3). However, grammar based testing was slightly better than CESE for `logictree`. `logictree`'s grammar allowed grammar based testing to explore words of higher height than CESE in 30 minutes. However, as we explore words with increasing height, there is a combinatorial blowup in the number of concrete words. If testing increased to one hour, CESE will explore words of greater height than traditional grammar testing.

Overall, CESE only performed 6% better than enumeration-based testing in the same budget. However, the number of inputs generated by CESE is usually an order of magnitude fewer than the number of inputs generated by exhaustive enumeration. Moreover, without test generation, the total run time of CESE inputs was 5 minutes, in contrast to 2.5 hours for exhaustive enumeration. Given that generated inputs are often added to the regression suite, this clearly indicates the superiority of CESE with respect to test suite quality.

Random Testing. We also applied random testing for 30 minutes. We fixed the input length to be 10 for each program because that value gave the best coverage results. As seen in Table 4, random testing explored the most inputs but was the least effective, because most random inputs were invalid and only exercised the syntax error handling code of the test programs. These results reaffirm that random testing is ineffective for programs requiring structured inputs.

Discussion and Limitations. As described in Section 2, both concolic execution and specification based testing have limitations. The combination of the two, as shown in our experiments, lessen these limitations, allowing CESE to have better performance and scale to larger programs.

Concolic testing is limited to the number and types of constraints generated by the program. If the constraints are beyond the theory of the constraint solver, concolic testing resorts to random testing. If the number of constraint becomes large, the constraint solver will become very slow. Although in our experiments all constraints were within the theory of `lp_solve`, both `lua` and `wuftpd` generated constraints that caused `lp_solve` to timeout when using naive concolic testing. Specifically, uses of `switch` statements and the `strlen` function introduces many inequalities in our constraints resulting in an exponential increase in the number of constraints to be solved. CESE greatly reduces the number of symbolic constants per input. Instead of testing the program with one large symbolic input, CESE divides the search space using knowledge of the grammar, thus allowing CESE to run all experiments without causing the underlying constraint solver to timeout. However, these limitations still affect performance when CESE is used to generate large inputs.

With larger inputs, lex and yacc style symbolic grammars do not give us enough constraints. Consider the following program that calculates the 10th factorial in the `bc` language:

```
define f (x) {
  if (x<=1) return(1)
  return (f(x-1)*x)
}
f (10)
```

This 60 character input contains 30 tokens and 9 symbolic constants in our symbolic grammar for `bc`. Generating interesting inputs of this size is still infeasible. Enumerating all possible symbolic strings and executing them with a large number of symbolic constants would take far too many resources.

Also, the grammar does not capture semantic properties of the input. For example in `bc` and `lua`, we must rely solely on concolic execution to ensure only defined functions are being used, assigned variables are being read, input is type correct, etc. Other properties are not captured by either the grammar nor can be found by concolic execution. For example in `lua`, there is a large set of library functions such as "print" that can be called. These functions do not appear in the grammar, and calls to these functions are sufficiently deep making it hard if not impossible for concolic execution to realize them. To remedy this, one can either use a

more descriptive grammar such as one that captures semantic properties *i.e.,* `f (10)` is a valid expression only if `f` has been defined, or to focus on specific classes of bugs.

In summary, our preliminary data suggests that CESE is highly effective in quickly generating a small test suite that can match or outperform many other test generation algorithms, and can get close to coverage obtained by manual testing while investing in a relatively short testing budget. However, more expressive specifications are required in order to explore deeper parts of the program state space.

	CESE			Grammar	
Program	**Height**	**Len**	**Sym**	**Height**	**Len**
bc	3	10	2	3	10
logictree	6	5	3	7	6
cuetools	11	40	3	11	40
lua	11	20	5	7	10
wuftpd	15	21	4	13	16

Table 5: Inputs: CESE and Grammar Based Testing. Height is the maximum applications of production rules and Len is the maximum generated input length for grammar based testing and CESE. Sym is the maximum number of symbolic constants in CESE inputs.

4.2 Bug Finding

Next we investigate the effectiveness of using CESE to find a specific class of memory access bugs. We use CESE to find a known buffer overflow in `wuftpd` that is difficult to find with Cute. We show that CESE allows concolic testing to scale so it can find interesting bugs that are beyond conventional techniques.

In the path lookup code in `wuftpd`, the `fb_realpath()` function has an off-by-one error that can be used as a buffer overflow with specifically crafted instructions. Figure 2 shows the bug. If `resolved` is equal to a non-root directory, then an extra "/" is added. Therefore, the `MAXPATHLEN` check is incorrect because `rootd` should be `!rootd` in line 07. Although this function is called by any command that uses pathnames, finding this bug is difficult because one needs to call this function with a pathname containing directory symlinks that results in a resolved pathname of exactly `MAXPATHLEN` size.

However even if we avoid this difficulty by restricting pathnames to contain a specific directory symlink that can exercise this error, finding the other pieces is still difficult. Suppose `MAXPATHLEN` is 1024 bytes and the directory link expands from a single letter directory link to a 23 letter directory name. Then the size of the string acting as the buffer must be exactly 1000 characters long. Also, there is the requirement of generating the right command. Random testing fails because the chance of both the directory and command string being generated is infinitesimally small. Although grammar-based testing will find the right commands to execute, grammar-based testing also fails because the number of valid strings smaller than the bug causing input is huge. For example, there are 27^{1000} strings of lower case letters and the character '/' with length 1000. If we use a combination of random and grammar-based testing,

```
/*
 * Join the two strings together, ensuring that the
 * right thing happens if the last component is
 * empty, or the dirname is root.
 */
00    if (resolved[0] == '/' && resolved[1] == '\0')
01        rootd = 1;
02    else
03        rootd = 0;
04
05    if (*wbuf) {
06        if (strlen(resolved) + strlen(wbuf) +
07           rootd + 1 > MAXPATHLEN) {
08            errno = ENAMETOOLONG;
09            goto err1;
10        }
11        if (rootd == 0)
12            (void) strcat(resolved, "/");
13        (void) strcat(resolved, wbuf);
14    }
```

Figure 2: `wuftpd` buffer overflow bug. In line 07, `rootd` in the comparison with `MAXPATHLEN` should be `!rootd`

we still are likely to fail to generate such a large string of that specific size.

Concolic testing can theoretically find this bug by using symbolic execution on string lengths. We experimentally compare pure concolic testing against CESE by using both techniques on `wuftpd`. The directory symlink is incorporated into both techniques as extra constraints on the input. The goal of the test is to generate any of the eight commands that can hit the bug and a string which, combined with the resolution of a known directory symlink, has a length of exactly `MAXPATHLEN`.

Length Abstraction. Although generating an input that exercises this buffer overflow requires a large pathname and therefore a large input buffer, both CESE and Cute can use a *length abstraction* for the strings while generating inputs [13]. The concolic execution input is changed to include both the original input buffer and a symbolic length. Concolic execution can track the length constraints on the inputs by by instrumenting the various string and memory manipulation library functions with the appropriate symbolic operations.

Using this length abstraction, CESE finds the overflow within four minutes while Cute does not find it within thirteen hours. Symbolic grammars allowed concolic execution to scale by reducing the number of constraints needed to be solved and the total number of inputs needed to be explored. Specifically, Cute is stuck tracking constraints in the parsing of commands. Even after 13 hours of execution and over 29,116 generated inputs, Cute still does not increase its coverage of `wuftpd` and does not find the bug. On the other hand, CESE gets the command from the grammar and thus has fewer symbolic values to solve and less inputs to generate. Therefore, CESE finds the bug in 4 minutes and achieves 43% branch coverage in 30 minutes while Cute gets only 19% coverage and does not find the bug within 13 hours.

5. RELATED WORK

There is a substantial body of related work in automatic test generation that leverage techniques in static analysis and grammar-based specification.

Program	LOC	CESE (30 min) Coverage	Inputs	Manual Testing Coverage
bc	12K	1010/2500 = 40%	133996	1235/2500 = 49%
logictree	8K	599/1376 = 43%	16827	740/1376 = 53%
cuetools	10K	572/1876 = 31%	99367	514/1876 = 27%
lua	32K	704/2422 = 29%	1061	1300/2422 = 54%
wuftpd	36K	552/1285 = 43%	10168	n/a
Total Coverage		3437/9459 = 36%		3789/8174 = 46%

Table 2: Coverage: CESE and Manual Testing. LOC is lines of code. Coverage is the branch coverage – executed branches divided by total branches. Inputs is the number of inputs generated. Manual Testing Coverage denotes the branch coverage for the developers' testcases. n/a denotes we could not find the developers' testcases.

Program	LOC	Cute (30 min) Coverage	Inputs	Cute (5 hour) Coverage	Inputs
bc	12K	865/2500 = 35%	148868	883/2500 = 35%	949948
logictree	8K	298/1376 = 22%	225103	341/1376 = 25%	2133323
cuetools	10K	456/1876 = 24%	147915	456/1876 = 24%	720384
lua	32K	584/2422 = 24%	2734	668/2422 = 28%	22939
wuftpd	36K	238/1285 = 19%	1139	238/1285 = 19%	11195
Total Coverage		2441/9459 = 26%		2586/9459 = 27%	

Table 3: Coverage: Cute. LOC is lines of code. Coverage is the branch coverage – executed branches divided by total branches, and Inputs is the number of inputs generated, for 30 minute Cute runs and 5 hour Cute runs.

Random Testing. Random testing is a widely used automated test generation technique [18, 4, 14, 27, 28]. However the probability of using pure random testing to find inputs that cause bugs is small [29] and many inputs result in the same code coverage. Heuristics and user guidance can be used to generate "good" random inputs that find bugs and increase coverage. Eclat [30] infers software behavior by examining example tests and uses this inferred knowledge to filter random test inputs. Randoop [31] creates a sequence of randomly selected method calls whose arguments come from previously generated inputs with the help of user specified filters. JCrasher [10] randomly generates parameters to test methods, and uses additional heuristics to rank exceptions in the tested code. We avoid having to use heuristics by guiding the input generation through constraints generated by the code, thus guaranteeing non-redundant tests and systematic coverage.

Specification-based Testing. Specification-based testing [17] is used to generate valid input that random testing has little chance of creating. Specification-based testing has been used for a wide variety of applications such as the Java virtual machine [34], XML testing [2], and Haskell [7].

Bounded exhaustive specification testing has been implemented in tools such as Yagg [9], TestEra [20] and Korat [5]. Both Korat [5] and TestEra [20] are tools that use specifications to test Java data structures. Yagg [9] generates all strings of a grammar up to a given depth.

Exhaustive enumeration of all possible inputs is generally infeasible so randomized testing is combined with specification-based testing. [26, 34, 7] use *stochastic test-data generation* where input is described with a grammar but rules are annotated with probabilities. Further annotations can be added to restrict the expansion of certain terms such as specifying the depth of productions, adding a guard, or specifying a production rule direction. Languages that describe these grammars include Geno [22] and DGL [25].

Our tool can use any specification-testing technique that generates strings such as [25, 22, 35, 26, 34, 7, 9]. While prior work uses random testing on terms that are left unexpanded, we use concolic testing on these terms. By using concolic testing, we can hope to find inputs that are hard to hit at random.

Hybrid Approaches. Recent work has combined static analysis with test input generation to increase coverage and eliminate false positives. CnC [11] creates inputs that cause errors from constraints generated by static analysis. DSD [12] further reduces false positives in CnC by inferring invariants using dynamic analysis. EXE [6], DART [16], and Cute [32] further extend this by running symbolic execution with concrete execution. In all these techniques whenever a bug is suspected, an input is generated that will expose the bug. Unfortunately, exhaustive running the program both concretely and symbolically is very expensive. [24] interleaves random testing with concolic testing to reduce the burden of constraint solving. Our work is similar in that it helps performance by allowing Cute to explore certain parts of the input but instead of using random testing, we rely on specifications, drastically reducing the number of inputs to be explored and increasing the speed of execution of each input. Some recent approaches use function summaries to reduce path explosion [15]. These techniques are complementary to CESE, but all share the same design goal of allowing concolic execution to go beyond the initial parsing code.

Program	LOC	Grammar (30 min) Coverage	Inputs	Random (30 min) Coverage	Inputs
bc	12K	779/2500 = 31%	262773	626/2500 = 25%	401673
logictree	8K	620/1376 = 45%	272851	526/1376 = 38%	461524
cuetools	10K	425/1876 = 23%	256748	452/1876 = 25%	428672
lua	32K	650/2422 = 26%	245130	541/2422 = 22%	390404
wuftpd	36K	377/1285 = 29%	290356	68/1285 = 5%	489904
Total Coverage		2851/9459 = 30%		2213/9459 = 23%	

Table 4: Coverage: Grammar Based Testing and Random Testing. LOC is lines of code. Coverage is the branch coverage – executed branches divided by total branches, and Inputs is the number of inputs generated.

6. CONCLUSIONS

Although concolic execution, in theory, can exhaustive search all paths in a program, in practice, there is rarely enough resources to rely solely on it. Instead, we show that some knowledge of the input domain (encoded through symbolic grammars) can provide extra constraints that enable exploration of deeper and more interesting paths quickly. Executing all symbolic strings from a symbolic grammar with concolic execution gives coverage equivalent to executing all valid strings from the corresponding concrete grammar. Symbolic grammars can be automatically generated from concrete grammars that are widely used in real applications. While technically simple, the technique is highly effective compared to random testing, specification based testing, and naive concolic execution. Our implementation of CESE shows that the technique can scale to real programs and find deep bugs in software that are difficult to find with other techniques. In fact, the failure to find the ftp buffer overflow bug with earlier techniques was the motivation for our work.

7. REFERENCES

[1] M. Berkelaar, J. Dirks, K. Eikland, and P. Notebaert. lp_solve (5.5.0.10). 2007.

[2] A. Bertolino, J. Gao, E. Marchetti, and A. Polini. Systematic generation of XML instances to test complex software applications. In *RISE*, 2006.

[3] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *ICSE*, 2004.

[4] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *ISSTA*, 2002.

[6] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler. EXE: automatically generating inputs of death. In *CCS*, 2006.

[7] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, 2000.

[8] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2:215–222, 1976.

[9] D. Coppit and J. Lian. yagg: an easy-to-use generator for structured test inputs. In *ASE*, 2005.

[10] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software Practice & Experience*, 34(11):1025–1050, 2004.

[11] C. Csallner and Y. Smaragdakis. Check 'n' Crash: combining static checking and testing. In *ICSE*, 2005.

[12] C. Csallner and Y. Smaragdakis. DSD-Crasher: a hybrid analysis tool for bug finding. In *ISSTA*, 2006.

[13] N. Dor, M. Rodeh, and S. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in c. In *PLDI*, 2003.

[14] J.E. Forrester and B.P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *4th USENIX Windows System Symposium*, 2000.

[15] P. Godefroid. Compositional dynamic test generation. In *POPL*, 2007.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[17] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Eng.*, 1(2):156–173, 1975.

[18] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[19] S.C. Johnson. YACC – yet another compiler-compiler. *Bell Labs Tehnical Report*, (32), 1975.

[20] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004.

[21] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[22] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *TestCom*, 2006.

[23] M.E. Lesk and E. Schmidt. Lex – a lexical analyser generator. *Bell Labs Tehnical Report*, (39), 1975.

[24] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[25] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990.

[26] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[27] B. P. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of macos applications using random testing. In *Random Testing*, 2006.

[28] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.

[29] A.J. Offutt and J.H. Hayes. A semantic model of program faults. In *ISSTA*, 1996.

[30] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, 2005.

[31] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, 2007.

[32] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *FSE*, 2005.

[33] M. Sipser. Introduction to the theory of computation. pages 91–101, 1997.

[34] E. Sirer and B. N. Bershad. Using production grammars in software testing. In *DSL*, 1999.

[35] D. R. Slutz. Massive stochastic testing of SQL. In *VLDB*, 1998.

[36] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, 2004.