Evaluating Interaction Patterns in Configurable Software Systems

Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter
Computer Science Department, University of Maryland, College Park, MD
Email: {elnatan,csfalcon,kkma,jfoster,aporter}@cs.umd.edu

Abstract

Many modern software systems are designed to be highly configurable, which makes testing them a challenge. One popular approach is combinatorial configuration testing, which, given an interaction strength t, computes a set of configurations to test such that all t-way combinations of option settings appear at least once. Basically, this approach assumes that interactions are complete in the sense that any combination of t options can interact and therefore must be tested. We conjecture, however, that in practical systems interactions are limited. If our conjecture is true, then new techniques might be developed to identify or approximate infeasible interactions, greatly reducing the number of configurations that must be tested. We evaluated this conjecture with an initial empirical study of several configurable software systems. In this study we used symbolic evaluation to analyze how the settings of run-time configuration options affected a test suite's line coverage. Our results strongly suggest that for these subject programs, test suites and configuration options, at least at the level of line coverage, interactions between configuration options are not complete.

1. Introduction

Many modern software systems include numerous user-configurable options. For example, network servers typically let users configure the active port, the maximum number of connections, what commands are available, and so on. While this flexibility helps make software systems extensible, portable, and achieve good quality of service, it can often yield an enormous number of possible system configurations. Moreover, failures can and do manifest themselves in some configurations but not in others, and so configurability can greatly magnify testing obligations. We call this the *software configuration space explosion* problem.

Researchers and practitioners have developed several strategies to cope with this problem. One popular approach is combinatorial testing [1], [2], [3], [4], which, given an *interaction strength* t, computes a *covering array*, a small set of configurations such that all possible t-way combinations of option settings appear in at least one configuration. The subject program is then tested under each configuration in the covering array, which will have very few configurations compared to the full configuration space of the program.

Several studies to date suggest that even low interaction strength (2- or 3-way) covering array testing can yield good line coverage while higher strengths may be needed for edge or path coverage or fault detection [2], [5], [6]. However, as far as we are aware, all of these studies have taken a black box approach to understanding covering array performance. Thus it is unclear how and why covering arrays work. On the one hand, a t-way covering array contains all possible t-way interactions, but not all combinations of options may be needed for a given program or test suite. On the other hand, a t-way covering array must contain many combinations of more than t options, making it difficult to tell whether t-way interactions, or larger ones, are responsible for a given covering array's coverage. We wish to obtain a better understanding of what level of interaction, and what specific interactions, truly control configuration spaces.

We conjecture that in practice configuration options often have quite incomplete interaction patterns. That is, we think that software systems are often structured in such a way that different options or groups of options interact only for some settings, if they interact at all. If true and if we can identify or approximate a system's actual interaction patterns, then testing and analysis effort could be directed in less expensive and/or more effective ways, saving a great deal of time and money throughout the software industry.

In this paper, we perform a whitebox investigation of the configuration spaces of three configurable software systems: vsftpd, ngIRCd, and grep. Our study relies on symbolic evaluation [7], [8], [9], which allows us to introduce *symbolic values* into a program and then track how they affect program execution. A symbolic value represents an unknown that can take on any value. When the symbolic evaluator encounters a branch that depends on a symbolic value, it conceptually forks execution and explores both possible branches.

In our study, we marked the initial values of selected run-time configuration options as symbolic, and then we ran test suites for the subject programs under Otter, a symbolic evaluator we developed. As Otter runs it also records line coverage information. We opted to measure line coverage because it is a simple and popular metric. Using Otter we were able to exhaustively calculate all possible program paths, for all possible settings of selected configuration options for these programs. This would have been impossible had we naively enumerated all configurations. We also generated 1-, 2-, and 3-way covering arrays for our subject programs and ran the test suites under those configurations.

Using this data, we discovered a number of interesting results about configuration options and line coverage for our subject programs. To determine how many configurations were actually necessary for maximum coverage, we used the symbolic evaluator results to compute a minimal set of configurations that yield the same coverage as all runs. We found that the sizes of these sets were relatively small—9 for vsftpd, 18 for ngIRCd, and 10 for grep. These sets are significantly smaller than the 3-way covering arrays for the same programs (41 for vsftpd, 131 for ngIRCd, 42 for grep), and yet they achieve slightly more coverage for vsftpd and grep. This suggests covering arrays are testing both more and fewer configurations than they need for maximum line coverage.

Investigating this gap further, we discovered that, for each program, there were some option settings that, while valid, would mask the effects of other options. For example, if the show_version option of grep is set, the other options are ignored, and grep exits after printing version information. Finding this kind of information typically requires insight into the program, but in this case we discovered it via the symbolic evaluator.

Next, to investigate interactions hidden by masking effects, we fixed the values of certain options to prevent the masking and ran the test suites under newly generated covering arrays. This time we found that, for all three programs, 3-way covering arrays yielded full line coverage. We also recomputed the minimal covering sets with the same options fixed, and found that the minimal covering sets were still smaller than the covering arrays.

Finally, we used Otter to discover what lines of code are guaranteed to be covered under certain combinations of option settings. For example, if a and b are options, we can compute what lines will always be covered if a=0 and b=2. Based on this information we examined if and how configuration options actually interact in our subject programs. We found that relatively few options interact and, where they do, those interactions often do not involve all possible values of the options. As a result, exercising all combinations of all t-tuples of configuration options is unnecessary for maximizing coverage.

In summary, our results strongly support our main hypothesis—that in practical systems, interaction among configuration options is not complete.

2. Configurable Software Systems

For our purposes, a configurable system is a generic code base and a set of mechanisms for implementing pre-planned variations in the system's structure and behavior. In practice, these variations are wide-ranging, including choices among hardware and operating system platforms (e.g., Windows vs Linux), software versions (e.g., which version of a source code file to include in a system), run-time features (e.g., enable/disable debugging output), among others. In this paper, we limit our attention to configuration options selected at run time, e.g., in configuration files or as command-line parameters.

Figure 1 illustrates several ways that run-time configuration options can be used, and explains why understanding their usage requires fairly sophisticated technology. All of these examples come from our experimental subject programs, which are written in C. In this figure, variables containing configuration options are shown in boldface.

The example in Figure 1(a) shows a section of vsftpd's command loop, which receives a command and then uses a long sequence of conditionals to interpret the command and carry out the appropriate action. The example shows two such conditionals that also depend on configuration options (all of which begin with tunable_ in vsftpd). In this case, the configuration options enable certain commands, and the enabling condition can either be simply the current setting of the option (as on lines 1–2) or may involve an interaction between multiple options (as on lines 6–7).

Not all options need be booleans, of course. Figure 1(b) shows an example from ngIRCd, in which the option Conf_MaxJoins is an integer that, if positive (line 13), specifies the maximum number of channels a

```
 1 ... else if (tunable_pasv_enable &&
 2             str_equal_text(&p_sess−>ftp_cmd_str, "EPSV"))
 3 {
 4   handle_pasv(p_sess, 1);
 5 }
 6 ... else if (tunable_write_enable &&
 7          (tunable_anon_mkdir_write_enable ||
 8           !p_sess−>is_anonymous) &&
 9          (str_equal_text(&p_sess−>ftp_cmd_str, "MKD") ||
10           str_equal_text(&p_sess−>ftp_cmd_str, "XMKD")))
11   handle_mkd(p_sess);
12 }
```

(a) Boolean configuration options (vsftpd)

```
13 if ((Conf_MaxJoins > 0) &&
14     (Channel_CountForUser(Client) >= Conf_MaxJoins))
15   return IRC_WriteStrClient(Client,
16                             ERR_TOOMANYCHANNELS_MSG,
17                             Client_ID(Client), channame);
```

(b) Integer-valued configuration options (ngIRCd)

```
18 else if(Conf_OperCanMode) {
19   /* IRC−Operators can use MODE as well */
20   if (Client_OperByMe(Origin)) {
21     modeok = true;
22     if (Conf_OperServerMode)
23       use_servermode = true; /* Change Origin to Server */
24   }
25 }
26 ...
27   if (use_servermode)
28     Origin = Client_ThisServer();
```

(c) Nested conditionals (ngIRCd)

```
29 not_text =
30   (((binary_files == BINARY_BINARY_FILES && !out_quiet)
31      || binary_files == WITHOUT_MATCH_BINARY_FILES)
32      && memchr (bufbeg, eol ? '\0' : '\200', buflim − bufbeg));
33 if (not_text &&
34     binary_files == WITHOUT_MATCH_BINARY_FILES)
35   return 0;
36 done_on_match += not_text;
37 out_quiet += not_text;
```

(d) Options being passed through the program (grep)

Figure 1. Example uses of configuration variables (bolded) in subjects.

user can join (line 14). In this example, error processing occurs if the user tries to join too many channels.

Figure 1(c) shows a different example in which two configuration options are tested in nested conditionals. This illustrates that it is insufficient to look at tests of configuration options in isolation; we also need to understand how they may interact based on the program's structure. Moreover, in this example, if both options are enabled then use_servermode is set on line 23, and its value is then tested on line 27. This shows that the values of configuration options can be indirectly carried through the state of the program.

Figure 1(d) shows another example in which configuration options are used indirectly. Here not_text is assigned the result of a complex test involving configuration options, and is then used in the conditional (lines 33–34) and to change the current setting of two other configuration options (lines 36–37).

Definitions. We define a *configuration* as a set $\{\ (V_1, C_1), (V_2, C_2), \ldots, (V_N, C_N)\ \}$, where each V_i is an option and C_i is its value, drawn from the allowable settings of V_i. In practice not all configurations make sense, e.g., feature X is not supported under option Y, and in this case we say there is an *inter-option constraint* between X and Y.

We say that a set of configuration options $V_1, \ldots, V_t$ *interact* if some behavior only occurs if the V_i's take specific settings. For purposes of our study, the behavior we are interested in is a set of lines being executed under a configuration. We say t options interact *strongly* if they interact for all possible settings; otherwise, they interact *weakly* if they interact for at least one setting, but not all.

3. Symbolic Evaluation

To understand how configuration options interact, we have to capture their effect on a system's runtime behavior. As we saw above, configuration options can be used in quite complex ways, and so simple approaches such as searching through code for option names will be insufficient. Instead, we use symbolic evaluation to capture all execution paths a program can take under any configuration.

The idea of enhancing testing with symbolic evaluation has been around for more than 30 years [7]. Recent advances in Satisfiability Modulo Theory (SMT) solvers, however, have enabled the technology to scale to practical problems.

Our symbolic evaluator, Otter,[1] is essentially a C source code interpreter, with one key difference. We allow the programmer to designate some values as *symbolic*, meaning they represent unknowns that may take on any value. Otter tracks these values as they flow through the program, and conceptually forks execution if a conditional depends on a symbolic value. Thus, if it runs to completion, Otter will simulate all paths

1. DART [8] and EXE [10] are two well known symbolic evaluators. By coincidence, Dart and Exe are the names of two rivers in Devon, England. The others are the Otter, the Tamar, the Taw, the Teign, and the Torridge.

through the program that are reachable for any values that the symbolic data can take. The key insight is that the number of paths executed is based on the branches in the program source code, rather than the much larger space of possible values for symbolic data.

To illustrate how Otter works, consider the example C source code in Figure 2(a). This program includes five variables that are inputs to the program: a, b, c, d, and input. The first four are intended to represent run-time configuration options, and so we initialize them on lines 1–2 with *symbolic values* α, β, γ, and δ, respectively. (In the implementation, the content of a variable v is made symbolic with a special call __SYMBOLIC(&v).) The last variable, input, is intended to represent program inputs other than configuration options. Thus we leave it as concrete, and it must be supplied by the user (e.g., as part of argv (not shown)).

The program continues by initializing local variable x (line 4) and then entering a series of conditionals and assignment statements. We have indicated five lines, numbered 1–5, whose coverage we are interested in. The execution path taken, and consequently which of those five lines are covered, depends on the settings of the symbolic values and concrete inputs.

Given a setting for input, Otter will exercise all the execution paths that are possible for any values of α, β, γ, and δ. Figure 2(b) shows these sets of paths as *execution trees* for two concrete "test cases" for this program: the tree for input=1 is on the left, and the tree for input=0 is on the right. Nodes in these graphs correspond to statements in the program, and branches represent places where Otter has a choice and hence "forks," exploring both possible paths.

For example, consider the tree with input=1. All executions begin by setting x to 0 and then testing the value of a, which at this program point contains α. Since there are no constraints on α, both branches are possible. For the sake of simplicity we will assume below that α and the other symbolic values may only represent 0 and 1, but Otter fully models symbolic integers as arbitrary 32-bit quantities.

Otter then splits its execution at the test of a, or, more precisely, it makes a choice and explores one branch, and then comes back and tries the other. First it assumes that α is true and reaches statement 1 (shown as the left branch). It then falls through to line 14 (the assignment to y) and performs the test on line 15 (x && input). This test is false, since x was set to 0 on line 4, hence there is no branch. We label this path through the execution tree as (A).

Notice that as we explored path (A), we made some decisions about the settings of symbolic values, specifically that α is true. We call this and any other constraints placed on the symbolic values a *path condition*. In this case, path (A) covers statement 1, and so any configuration that sets a=1 on line 1 (corresponding to α being true), with arbitrary choices for the values of β, γ, and δ, will cover statement 1. This is what makes symbolic evaluation so powerful: With a single predicate we characterized the behavior of many possible concrete choices of symbolic inputs (in this case, there would be 2^3 possibilities for all combinations of b, c, and d).

Otter continues by returning to the last place it forked and trying to explore the other path. In this case, it returns to the conditional on line 5, assumes α is false by adding $\neg\alpha$ to the path condition, and continues exploring the execution tree. Each time Otter encounters a conditional, it actually calls an SMT solver to determine which branches of the conditional are possible based on the current path condition. Let p be the current path condition, and suppose Otter reaches a branch with guard g. Then if $p \wedge g$ is satisfiable, Otter explores the true branch, and if $p \wedge \neg g$ is satisfiable, it explores the false branch. Execution splits if both are satisfiable. Otter continues in this manner until it has explored all possible paths. In theory this might not terminate in any reasonable amount of time, but in our experiments, we were able to achieve termination even when setting many configuration options as symbolic, perhaps because configuration options tend to be used in fairly restricted ways.

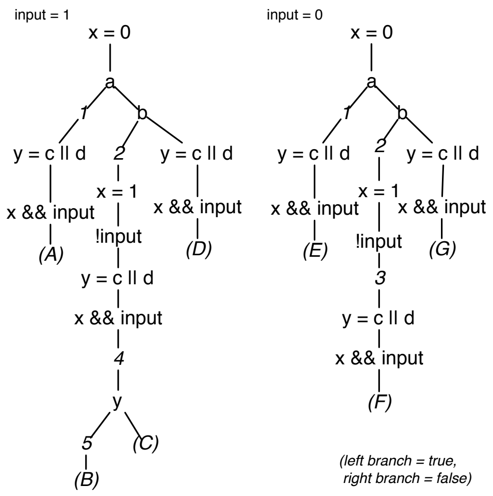
There are a few other interesting things to notice about these execution trees. First, consider the execution paths labeled (B) and (C). Notice that because we have chosen β to be true on this path, we set x=1, and hence x && input is true, allowing us to reach statements 4 and 5. This is analogous to the example in Figure 1(c), in which a configuration option choice resulted in a change to the program state (setting x=1) that allowed us to cover some additional code.

Also, notice that if input=1, there is no way to reach statement 3, no matter how we set the symbolic values. Hence coverage depends on choices of both symbolic values and concrete inputs.

In total, there are four paths that can be explored when input=1, and three paths when input=0. However, there are 2^4 possible assignments to the symbolic values α, β, γ, and δ. Hence using symbolic evaluation for these test cases enables us to gather full coverage information with only 7 paths, rather than the 32 runs required if we had tried all possible combinations of symbolic and concrete inputs. This is what makes the results in this paper even possible—we can effectively get the same result as if we had tried all possible combinations of configuration options with far fewer

```
int a=α, b=β,
    c=γ, d=δ; /* symbolic */
int input=...; /* concrete */
int x = 0;
if (a)
  /* 1 */
else if (b) {
  /* 2 */
  x = 1;
  if (!input) {
    /* 3 */
  }
}
int y = c || d;
if (x && input) {
  /* 4 */
  if (y)
    /* 5 */
}
```

(a) Example program



(b) Full execution trees

	Path condition and config.	Cov
input = 1	(A) α a $=1$	1
	(B) $\neg\alpha \wedge \beta \wedge (\gamma \vee \delta)$ a $=$ d $= 0,$ b $=$ c $= 1$	$2, 4, 5$
	(C) $\neg\alpha \wedge \beta \wedge \neg(\gamma \vee \delta)$ a $=$ c $=$ d $= 0,$ b $= 1$	$2, 4$
	(D) $\neg\alpha \wedge \neg\beta$ a $=$ b $= 0$	$-$
input = 0	(E) α a $= 1$	1
	(F) $\neg\alpha \wedge \beta$ a $= 0,$ b $= 1$	3
	(G) $\neg\alpha \wedge \neg\beta$ a $=$ b $= 0$	$-$

(c) Path conditions and configurations

Figure 2. Example symbolic evaluation.

paths than that would entail if done concretely.

3.1. Minimal Covering Sets

One of our basic hypotheses is that many options do not interact, and we can still get good coverage even if we run our test cases under fewer configurations than suggested by covering arrays. As we just saw, given a set of symbolic values for configuration options, Otter finds *all* possible program paths that can be visited by setting those options. Thus, we can test our hypothesis by using Otter to find a minimal set of configurations such that, if we run all the test cases under just those configurations, we will achieve the same coverage as the full set of configurations. We call such a set a *minimal covering set*, and if it is small, this will lend support to our hypothesis.

For example, Figure 2(c) summarizes the path conditions for all seven paths of our example program, gives an initial configuration that will yield that path (i.e., that satisfies the path condition), and lists which (specially marked) lines each path covers. Since this example is small, we can easily find a minimal covering set: Condition (F) is the only one that covers statement 3, so we need a configuration that satisfies it. If we pick a configuration that also satisfies (B), then we can cover lines 2–5 with just that one configuration. In this case, we can set a=0, b=c=1, and d=anything. Now only statement 1 remains to be covered, which we can do by satisfying path condition (A) (which is the same as (E)). In this case, we can set a to 1, and all other options to anything. Thus, here is one minimal covering set:

Config	Paths	Coverage
a $=$ d $= 0,$ b $=$ c $= 1$	$(B), (F)$	$2, 3, 4, 5$
a $= 1,$ b $=$ c $=$ d $= 0$	$(A), (E)$	1

This is a simple example, but finding a precise minimal covering set is intractable in general, for two reasons. First, above we determined (B)'s and (F)'s path conditions were simultaneously satisfiable, and hence those paths could be covered by one configuration. Scaling this up to all sets of path conditions would require numerous calls to the SMT solver, which would likely be computationally impractical. Second, computing an optimal covering set is NP-hard [11], and our subject programs have a multitude of paths.

Instead we compute an approximate answer that may be larger than the actual minimal covering set. Our algorithm begins by calling the SMT solver on each path condition to find a partial configuration satisfying that condition. The partial configuration contains settings only for options referenced in the path conditions, and the other options are omitted (as in Figure 2(c)).

Next, we process the list of partial configurations, checking if each one is *compatible* with any other seen so far. Two partial configurations are compatible if they assign the same values to options on which they overlap. We merge compatible configurations (replace each by the union of their settings) and record their coverage as the union of the coverage of the original configurations. Note this may be suboptimal, both because we merge greedily and because two configurations may be incompatible even if their underlying path conditions are simultaneously satisfiable.

Finally, we use a greedy algorithm to search the combined configurations for a covering set. We begin by picking a configuration that covers the largest number of lines (ties broken arbitrarily). Then we pick a configuration that covers the largest number of lines that have not been covered so far, and so on until we achieve full coverage. For example, using the configurations in Figure 2(c), the following chart shows one sequence our algorithm may pick if (C)'s and (F)'s configurations were merged.

Path	Config	Covered so far
(B)	$\mathsf{a} = \mathsf{d} = 0, \mathsf{b} = \mathsf{c} = 1$	$\{2, 4, 5\}$
$(A), (E)$	$\mathsf{a} = 1$	$\{1, 2, 4, 5\}$
$(C), (F)$	$\mathsf{a} = \mathsf{c} = \mathsf{d} = 0, \mathsf{b} = 1$	$\{1, 2, 3, 4, 5\}$

Notice that our algorithm determines that three configurations are necessary for full coverage, whereas the optimal solution requires only two configurations. Nevertheless, we have found our approach yields small covering sets in practice.

3.2. Guaranteed coverage

Beyond finding a minimal covering set, we also use Otter to compute the *guaranteed coverage* of a predicate. We define $\mathit{Cov}(p)$ to be the set of lines always covered in an execution whose symbolic values satisfy p. From this, we can discover useful information about configuration option interactions.

For example, $S_0 = \mathit{Cov}(\mathit{true})$ is the set of lines covered in all program executions, regardless of the configuration options. Similarly, if a program includes only boolean-valued options x_i, then $S_1 = \bigcup_i (\mathit{Cov}(x_i) \cup \mathit{Cov}(\neg x_i))$ is the set of lines guaranteed covered if we pick all 1-way covering arrays or *all values* of the options. We can go still further, and define

$$S_2 = \bigcup_{i \neq j} (\mathit{Cov}(x_i \wedge x_j) \cup \mathit{Cov}(x_i \wedge \neg x_j) \cup \mathit{Cov}(\neg x_i \wedge x_j) \cup \mathit{Cov}(\neg x_i \wedge \neg x_j))$$

to be the set of lines guaranteed covered by all 2-way covering arrays, and so on. Thus, using $\mathit{Cov}(p)$, we can distinguish what is guaranteed to be covered by t-way interactions in a covering array, and what is covered by happenstance.

We can also use $\mathit{Cov}(p)$ to find interactions among configuration options. Let x_1 and x_2 be symbolic values used to initialize two options. Then if $\mathit{Cov}(x_1 \wedge x_2)$ is a strict superset of $(\mathit{Cov}(x_1) \cup \mathit{Cov}(x_2))$, there is some code that is only covered if both x_1 and x_2 are true, and thus we can conclude that x_1 and x_2 interact.

We compute $\mathit{Cov}(p)$ by first finding $\mathit{Cov}_T(p)$, the lines guaranteed to be hit under p by test case T, for each test case; then, $\mathit{Cov}(p) = \bigcup_T \mathit{Cov}_T(p)$. To compute $\mathit{Cov}_T(p)$, let p_i be the path conditions from T's symbolic evaluation, and let $L(p_i)$ be the lines that occur in that path. Then $\mathit{Cov}_T(p)$ is

$$\begin{aligned} \mathit{Compat}(p) &= \{p_i \mid \mathit{SAT}(p_i \wedge p)\} \\ \mathit{Cov}_T(p) &= \bigcap_{p_j \in \mathit{Compat}(p)} L(p_j) \end{aligned}$$

In other words, first we compute the set of predicates p_i such that p and p_i are simultaneously satisfiable. If this holds for p_i, the lines in $L(p_i)$ *may* be executed if p is true. Since our symbolic evaluator explores all possible program paths, the intersection of these sets for all such p_i is the set of lines guaranteed to be covered if p is true.

Continuing with our running example, here is the coverage guaranteed by some predicates:

p	$\mathit{Compat}(p)$ ($\mathsf{input} = 1$)	$\mathit{Compat}(p)$ ($\mathsf{input} = 0$)	$\mathit{Cov}(p)$
α	(A)	(E)	$\{1\}$
$\neg\alpha$	$(B), (C), (D)$	$(F), (G)$	$\emptyset$
$\neg\alpha \wedge \beta$	$(B), (C)$	(F)	$\{2, 3, 4\}$
$\neg\alpha \wedge \beta \wedge \gamma$	(B)	(F)	$\{2, 3, 4, 5\}$

Note that we cannot guarantee covering line 5 without setting three symbolic values (although we could have picked δ instead of γ).

As with our algorithm for minimal covering sets, our computation of $\mathit{Cov}(p)$ is an approximation. The precision of this approximation warrants further study, but based on manual examination for our data, it appears accurate and gives us considerable insight into the configuration spaces of our subjects.

3.3. Implementation

Otter is written in OCaml, and it uses CIL [12] as a front end to parse C programs and transform them into an easier-to-use intermediate representation.

The general approach used by Otter mimics KLEE [9]. A symbolic value in Otter represents a sequence of untyped bits, e.g., a 32-bit symbolic integer is treated as a vector with 32 symbolic bits in Otter. This low-level representation is important because many C programs perform bit manipulations that must be modeled accurately. When a symbolic expression has to be evaluated (e.g., at branches), Otter invokes STP [13], an SMT solver optimized for bit vectors and arrays.

Otter supports all the features of C we found necessary for our subject programs, including pointer arithmetic, function pointers, variadic functions, and type casts. Otter currently does not handle multi-threading, dereferencing symbolic pointer values, floating-point

	vsftpd	ngIRCd	grep
Version	2.0.7	0.12.0	2.4.2
LoC (sloccount)	10,482	13,601	8,456
LoC (executable)	4,384	4,421	3,498
# Test cases	58	59	70
# Analyzed conf. opts.	47	13	19
Boolean	33	5	16
Integer	14	8	3
# Excluded conf. opts.	75	16	0

Figure 3. Subject program statistics.

arithmetic, or inline assembly. Multithreading is used in vsftpd's standalone mode and in ngIRCd, but we handle this by interpreting fork() as driving the program to the subprocess that provides the main functionality, and ignoring the other process (which performs little or no critical work for our purposes). The other unsupported features either do not appear in our subject programs or do not affect the results of our study; we leave the handling of these features as future work.

All of our subject programs interact with the operating system in some way. Thus, we developed "mock" libraries that simulate a file system, network, and other needed OS components. Our libraries also allow us to control the contents of files, data sent over the network, and so on as part of our test cases. Our mock library functions are mostly written in C and are executed by Otter just like any other program. For example, we simulate a file with a character array, and a file descriptor points to some file and keeps the current position at which the file is to be read or written.

4. Subject Programs

The subject programs for our study are vsftpd, a widely-used secure FTP daemon; ngIRCd, the "next generation IRC daemon"; and grep, a popular text search utility. All of our subject programs are written in C. Each has multiple configuration options that can be set either in system configuration files or through command-line parameters.

Figure 3 gives descriptive statistics for each subject program. We list each program's version number, source lines of code as computed by sloccount [14], and the number of executable lines of code in the CIL representation of the program. In our study, we measure coverage with respect to this latter count. Note that about 6% of the total executable lines of vsftpd are devoted to a two-process mode. However, as Otter does not support multiprocess symbolic evaluation, we forced our tests to run in single-process mode.

Next, we list the number of test cases. Vsftpd does not come with its own test suite, so we developed tests

	vsftpd	ngIRCd	grep
# Paths	107,456	22,904	270,393
Coverage (%)	49.2	62.2	64.7
# Examined opts/tot	20/47	13/13	16/19

Figure 4. Summary of symbolic evaluation.

to exercise functionality such as logging in; uploading, downloading, and renaming files; and asking for system information. ngIRCd also does not come with its own test suite, and we created tests based on the IRC functionality defined in RFCs 1459, 2812 and 2813. Our tests cover most of the client-server commands (e.g., client registration, channel join/part, messaging and queries) and a few of the server-server commands (e.g., connection establishment, state exchange). Grep comes with a test suite consisting of hundreds of tests. We symbolically evaluated these tests in Otter to determine their maximum line coverage. Then, without sacrificing line coverage, we selected a subset of test cases and did all further analyses on this sample.

Finally, we list the number of configuration options treated as symbolic, including the breakdown of boolean and integer options, and the number of configuration options left as concrete. Our decision to leave some options concrete was primarily driven by two criteria: whether the option was likely to expose meaningful behaviors and our desire to limit total analysis effort. This approach allowed us to run Otter numerous times on each program, to explore different scenarios, and to experiment with different kinds of analysis techniques. We used default values for the concrete configuration options, except the one used to force single-process mode in vsftpd. Grep includes a three-valued string option to control which functions execute the search; for simplicity, we introduced a three-valued *integer* configuration option exe_index and set the string based on this value.

5. Data and Analysis

To calculate the execution paths coverable by our subject programs, we ran each one's test cases in Otter, with the configuration options treated as symbolic as discussed above. To do this we used the Skoll system and infrastructure developed and housed at the Univ. of MD [15]. Skoll allows users to define configurable QA tasks and run them across large virtual computing grids. For this work we used ∼40 client machines to run our analyses. The final results reported here, including substantial post-processing, required about 5–6 days of elapsed time. From these runs we derived line coverage and other information, presented in Figure 4.

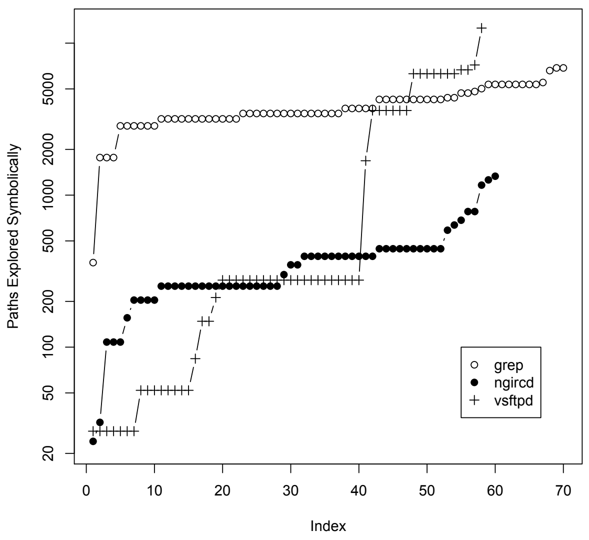


Figure 5. Number of paths per test case.

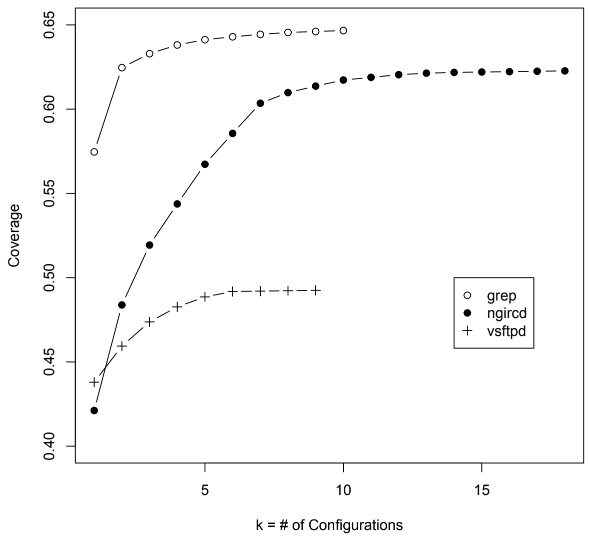


Figure 6. Coverage at each step of minimal covering set algorithm.

The first row of the figure counts the number of execution paths through these programs. While Otter found many thousands of paths, recall that these are actually *all* possible paths for any settings of the symbolic configuration options. Had we instead opted to naively run each test case under all possible configuration option combinations, it would have required 1.5×10^{15} executions for vsftpd, 15 million for ngIRCd, and 124 million for grep.

The second row of the figure shows the coverage cumulatively achieved by all these paths. In other words, this is the maximum coverage achievable for these test suites considering all possible configurations. The last row shows how many configuration options are actually evaluated in at least one path, compared to the total number of options. We can see that many vsftpd options are never evaluated in these runs, while all or almost all options for ngIRCd and grep are.

Figure 5 shows the number of execution paths broken down by individual test cases; the x-axis is sorted from the fewest to the most paths. Here we see that different test cases can exercise very different numbers of paths, though they appear to cluster into a handful of groups. On further examination we discovered that the clustering occurs because each cluster of test cases exercises code that depend on the same combinations of configuration options.

5.1. Minimal Covering Sets

Using the line coverage data, we computed a minimal covering set for each program. As mentioned earlier this is a (small) set of configurations such that, if we run the tests under them, we get the same coverage as if we run under all configurations.

Figure 6 graphs the progression of the greedy algorithm as it covers the lines of each subject program. The position value of the rightmost point on an arc indicates the minimal covering set's size. In total, vsftpd needed 9 configurations to achieve full coverage, ngIRCd needed 18, and grep needed 10. Clearly, these numbers are far smaller than the total size of the configuration space for each program.

We can see that each subject program follows the same general trend, with most coverage achieved early on. Grep is the most extreme example, getting much of its maximum possible coverage with the first two configurations. Investigating further, we found this occurs because grep has a lot of common code that runs in most configurations, and this code is covered early. The remaining configurations cover variations in grep's functionality that are localized in small sections of code. In ngIRCd and vsftpd, there are larger pieces of code that depend on configuration options, and so coverage increases more gradually. The first few configurations for ngIRCd and vsftpd drive tests to execute the normal paths in the program, which cover most of the code. In contrast, the last configurations for ngIRCd and vsftpd cover only a few new lines each, typically for paths that correspond to error conditions.

5.2. Comparison to Covering Arrays

Next, for each subject program we computed one t-way covering array for each value of $t \in \{1, 2, 3\}$.

	vsftpd		ngIRCd		grep	
	# configs, coverage (%)					
1-way	3	2.6	7	56.2	3	62.6
2-way	12	40.1	28	62.0	13	64.1
3-way	41	44.8	131	62.2	42	64.3
Min.	9	49.2	18	62.2	10	64.7

(a) Full configuration space

	vsftpd		ngIRCd		grep	
	# configs, coverage(%)					
1-way	3	43.5	7	61.5	3	63.2
2-way	12	48.9	28	61.6	12	64.0
3-way	40	48.9	119	61.6	37	64.2
Min.	6	48.9	8	61.6	10	64.2

(b) Refined configuration space

Figure 7. Coverage obtained with covering arrays.

To build the covering arrays we had to discretize the values of all integer-valued configuration options. This step normally requires developer input, either based on their knowledge of the system or their use of a structured analysis technique such as category-partition analysis [16]. As a surrogate for developer insight, we examined the path conditions that led to maximum coverage and derived concrete option settings consistent with the path conditions. This may be overly optimistic, in the sense that the symbolic evaluator could discover information that may not be known to the tester in practice.

We then ran Otter on each test case for each covering array configuration. This time, however, instead of making the configuration options symbolic we used the concrete settings dictated by the covering array. We could, of course, have done these runs outside of the symbolic evaluator, but this would have required us to construct separate instrumentation chains for concrete and symbolic evaluation.

Figure 7(a) summarizes coverage information for different strength covering arrays. On each line we list the number of configurations in the covering array and the line coverage. The last line repeats the minimal covering set information from Section 5.1 for comparison. We see that vsftpd has very poor coverage with the 1-way covering array, but its coverage increases sharply with the 2-way covering array. In contrast, ngIRCd and grep both start with high coverage and quickly get near maximal coverage with the 2-way covering array.

Examining this data, we can draw several initial observations. First, compared to minimal covering sets, the covering arrays for all 3 programs required running many more configurations than necessary to achieve maximum coverage. We also see that in several cases individual configurations in the covering array are redundant in the sense that they add no unique line coverage over other configurations in the covering array. Finally, these data also highlight the practical difficulty of knowing which strength t to use. For ngIRCd a 3-way covering array was enough, but for the other programs it was not.

5.3. Refining the Configuration Space

As we investigated the covering array behavior more closely, we found that our subject programs exhibited masking effects caused by specific configuration option settings. That is, for each program, a small number of options, set in particular ways, completely dictated program behavior, so that all other option values were effectively ignored.

One example in grep is the show_version option, which, when enabled, causes grep to display version information and then exit, making all other options irrelevant. Since covering arrays densely pack option settings into as few configurations as possible, this masking behavior can impede coverage.

Along similar lines, when vsftpd is in single-process mode (as it is in our runs), any configuration that enables local logins or SSL will encounter code that shuts down the system immediately. And in ngIRCd, three configuration options can be set in such a way that either clients have more difficulty connecting to the server, or that response timeouts occur frequently causing clients to drop their connections. These are not invalid configurations, but their behavior prevents full exploration of the program.

None of this creates problems for symbolic evaluation; Otter essentially works around these configurations to find all possible program paths. However, to give us a second point for comparison, we fixed the relevant settings in each program (2 option settings for grep, 3 for ngIRCd, and 3 for vsftpd) to prevent these masking effects.

We then conducted a second covering array analysis. Figure 7(b) details the coverage achievable under these new models. Not surprisingly, coverage is greatly improved at lower strengths because coverage due to 3 or more options in the original model is now achievable with 2 or 3 fewer options. Note that the maximum coverage is slightly lower than before, because some lines are now unreachable due to the option settings. Vsftpd and ngIRCd now reach maximum coverage with 2-way covering arrays, while grep now does so with a 3-way covering array. We also recomputed the minimal coverage sets. Now only 6 configurations are needed for vsftpd, 8 for ngIRCd, and 10 for grep.

	vsftpd		ngIRCd		grep	
	%	O	%	O	%	O
0-way	29.0	–	33.6	–	6.9	–
1-way	40.3	11	50.0	7	60.1	7
2-way	45.7	13	60.1	10	63.2	12
Max	48.9	20	61.6	13	64.2	16

% = coverage O = # options

Figure 8. Guaranteed coverage analyses.

Covering arrays sizes are thus still greater than those of the minimal covering sets, implying that not all option combinations are needed even for this new configuration space.

5.4. Understanding Interactions

To better understand which specific configuration options interact, we used the algorithm from Section 3.2 to compute the coverage guaranteed by various option settings in the refined configuration spaces. First, we computed $Cov(true)$, which we call *guaranteed 0-way coverage*. These are lines that are guaranteed to be covered for any choice of options. Then for every possible option setting $\mathsf{x} = \mathsf{v}$ from the covering arrays, we computed $Cov(\mathsf{x} = \mathsf{v})$. The union of these sets is the *guaranteed 1-way coverage*, and it captures what lines will definitely be covered by a 1-way covering array. Similarly, we also computed $Cov(\mathsf{x1} = \mathsf{v1} \wedge \mathsf{x2} = \mathsf{v2})$ for all possible pairs of option settings, which is *guaranteed 2-way coverage*.

Figure 8 presents the results. Note that higher-level guaranteed coverage always includes the lower level, e.g., if a line is covered no matter what the settings are (0-way), then it is certainly covered under particular settings (1- and 2-way). We also list, for 1- and 2-way covering arrays, the number of options involved in the new coverage at that level. The last line lists the maximum possible coverage (from the last line of Figure 7(b)) and the maximum number possible of configuration options (the count of touched options from Figure 4).

The figure shows that 1-way covering arrays guarantee a significant amount of line coverage, and that most lines are guaranteed covered by 2-way covering arrays. However, higher strength covering arrays are still required to guarantee full coverage. We confirmed through manual inspection that all our subject programs have lines that cannot be covered unless 3 or more options take specific settings. Notice that the actual n-way covering arrays (Figure 7(b)) achieved higher coverage than is guaranteed; in these cases, the additional lines were in essence covered by chance.

We can also use guaranteed coverage analysis to identify the specific options and settings that interact, at least to the level of 2-way interactions. Figure 9 represents grep's interactions. In this graph, an isolated node x=v means that at least one line is guaranteed to be covered when x=v, and that line was not guaranteed under 0-way coverage. If nodes x1=v1 and x2=v2 are connected by an edge, then the conjunction of those settings guarantees some line coverage not guaranteed under 1-way coverage. Lastly, whenever all possible settings of an option are equivalent, meaning they are connected to the same set of nodes in the graph, we replace them by a single node x=$*$.

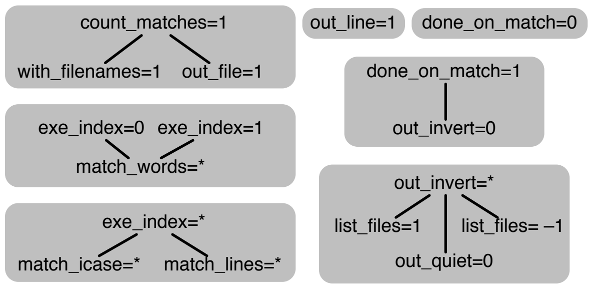


Figure 9. Interactions between config. options.

We can draw several observations from this graph. First, many options are missing from the graph because they guarantee no coverage. From the perspective of line coverage these options are irrelevant up to 2-way interactions. We also see that several options display weak interactions—they guarantee coverage for some settings, but others are irrelevant. For example, list_files=1 and list_files=-1 guarantee some coverage, but list_files=0 does not. Importantly, those options that do interact appear to form disjoint clusters. Essentially, rather than a monolithic entity, the configuration space of grep is the combination of multiple smaller configuration subspaces. Next, we observe a small number of strong interactions, such as those between exe_index and match_icase and between exe_index and match_lines. Although we have not fully confirmed this, it appears that the size of the minimal covering sets is in part driven by the number of combinations in strongly interacting options. Finally, we examine the specific lines guaranteed covered (analysis not shown), and find that in many cases the guaranteed coverage of one option combination subsumes that of another. Again, this fact partially explains why minimal covering sets are so much smaller than covering arrays.

5.5. Threats to Validity

For this work we selected 3 subject programs. Each is widely used, but small in comparison to some industrial applications. Our test suites taken together have reasonable, but not complete, line coverage. Individually the test cases tend to be focused on specific functionality, rather than combining multiple activities in a single test case. In that sense they are more like a typical regression suite than a customer acceptance suite. To compute the covering arrays, we had to discretize integer-valued options. We based the choice of test values on knowledge taken from our analysis of the code. This may overstate the effectiveness of the covering array approach in practice.

6. Related Work

Symbolic Evaluation. In the mid 1970's, King was one of the first to propose symbolic evaluation as an aid to program testing [7]. However, at that time theorem provers were much less powerful, limiting the approach's potential. Recent years have seen remarkable advances in Satisfiability Modulo Theory and SAT solvers, which has enabled symbolic evaluation to scale to practical problems.

Some recently developed symbolic evaluators include DART [8], [17], CUTE [18], SPLAT [19], EXE [10], and KLEE [9]. There are important technical differences between these systems, e.g., DART uses *concolic execution*, which mixes concrete and symbolic evaluation, and KLEE uses pure symbolic evaluation. However, at a high level, the basic idea is the same for all these tools: the programmer marks values as symbolic, and the evaluator explores all possible program paths reachable under arbitrary assignments to those symbolic values. As we mentioned earlier, Otter is closest in implementation terms to KLEE.

Software Engineering for Configurable Systems. Several researchers have explored using design of experiments (DoE) theory to generate, select or evaluate test suites. Mandl [3] first used orthogonal arrays, a special type of covering array in which all t-sets occur *exactly* once, to test enumerated types in ADA compiler software. This idea was extended by Brownlie *et al.* [2] who developed the orthogonal array testing system (OATS). Yilmaz et al. [20] applied covering arrays to test configurable systems. They showed that covering arrays were effective not only in detecting failures, but also in characterizing the specific failure inducing options.

Some of these researchers have empirically demonstrated that DoE techniques can be effective at fault detection and provide good line or path coverage. Dalal *et al.* [21] argued that testing all pairwise interactions in a software system can find many faults. In further work, Burr *et al.* [22], Dunietz *et al.* [5], and Kuhn *et al.* [6] provided more empirical results to show that DoE techniques can be effective. Dunietz *et al.* in particular showed that for a single subject system, low strength covering arrays provided code block coverage of the system, while higher strength arrays were needed to achieve high path coverage. Cohen et al. [23] created covering arrays for input data and then ran these over multiple configurations of a web browser. They found that changing some configuration parameters had little effect on block coverage for their test suite. With the possible exception of Cohen et al, these studies appear consistent with our findings, but since each study used black box techniques, none has commented on the code-level mechanisms accounting for the observed behavior.

7. Conclusions and Future Work

We have presented an initial experiment using symbolic evaluation to study the interactions among configuration options for three software systems. Keeping existing threats to validity in mind, we drew several conclusions. All of these conclusions are specific to our programs, test suites, and configuration spaces; further work is clearly needed to establish more general trends.

First, the minimal covering sets we computed were very small, much smaller in fact than 3-way covering arrays. Second, covering arrays produced very high line coverage, but they still lacked some configurations needed for maximum coverage and included many configurations that are redundant with respect to line coverage. Third, for up to 2-way interactions, we found that small groups of options sometimes interacted with each other, but not with options in other groups. Hence we conclude that the overall configuration spaces were not monolithic, but were actually a combination of disjoint configuration subspaces. Taken together these results strongly support our main hypothesis—that in practical systems, many configuration options simply do not interact with each other.

Based on this work, we plan to pursue several research directions. First we will extend our studies to better understand how configurability affects software development. Some initial issues we will tackle include increasing the number and types of options; repeating our study on more and larger subject systems; and expanding our guaranteed coverage analyses to higher strengths. We will also repeat our experiments using more complex coverage criteria such as edge, condition, and multiple criteria decision coverage.

We also plan to enhance our symbolic evaluator to improve performance, which should enable larger scale studies. One potential approach is to use path pruning heuristics to reduce the search space, although we would no longer have complete information.

Finally, we will explore potential applications of our approach and results. For example we may be able to use symbolic evaluation to discretize integer-valued configuration options and to identify option settings that exhibit masking effects. As our results show that different test cases depend on different configuration options, we will investigate how this information can be used to support a variety of software tasks, such as test prioritization, configuration-aware regression testing and impact analysis. Finally, we will investigate how guaranteed coverage information might be presented to developers in ways that help them better understand how configuration interactions influence specific lines of code.

8. Acknowledgments

We would like to thank Cristian Cadar and the other authors of KLEE for making their code available to us. We would also like to thank Mike Hicks for helpful comments. This research was supported in part by NSF CCF-0346982, CCF-0430118, CCF-0524036, CCF-0811284 and CCR-0205265.

References

[1] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: an approach to testing based on combinatorial design," *TSE*, vol. 23, no. 7, pp. 437–44, 1997.

[2] R. Brownlie, J. Prowse, and M. S. Phadke, "Robust testing of AT&T PMX/StarMAIL using OATS," AT&T Technical Journal, vol. 71, no. 3, pp. 41–7, 1992.

[3] R. Mandl, "Orthogonal Latin squares: an application of experiment design to compiler testing," *Commun. ACM*, vol. 28, no. 10, pp. 1054–1058, 1985.

[4] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *ICSE*, 2003, pp. 38–48.

[5] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. M. ws, and A. Iannino, "Applying design of experiments to software testing," in *ICSE*, 1997, pp. 205–215.

[6] D. Kuhn and M. Reilly, "An investigation of the applicability of design of experiments to software testing," in *NASA Goddard/IEEE Software Engineering Workshop*, 2002, pp. 91–95.

[7] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[8] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, 2005, pp. 213–223.

[9] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," in *CCS*, 2006, pp. 322–335.

[11] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*. Plenum Press, New York, 1972.

[12] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *International Conference on Compiler Construction*, 2002, pp. 213–228.

[13] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *CAV*, July 2007.

[14] D. Wheeler, "Sloccount," 2009. [Online]. Available: http://www.dwheeler.com/sloccount/

[15] A. Porter, C. Yilmaz, A. M. Memon, D. C. Schmidt, and B. Natarajan, "Skoll: A process and infrastructure for distributed continuous quality assurance," *TSE*, vol. 33, no. 8, pp. 510–525, August, 2007.

[16] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Commun. ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[17] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS*. Internet Society, 2008.

[18] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *FSE-13*, 2005, pp. 263–272.

[19] R.-G. Xu, P. Godefroid, and R. Majumdar, "Testing for buffer overflows with length abstraction," in *ISSTA*, 2008, pp. 27–38.

[20] C. Yilmaz, M. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *TSE*, vol. 31, no. 1, pp. 20–34, 2006.

[21] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in *ICSE*, 1999, pp. 285–294.

[22] K. Burr and W. Young, "Combinatorial test techniques: Table-based automation, test generation and code coverage," in *ICSE Analysis & Review*, 1998.

[23] M. B. Cohen, J. Snyder, and G. Rothermel, "Testing across configurations: implications for combinatorial testing," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 6, pp. 1–9, 2006.