iGen: Dynamic Interaction Inference for Configurable Software

ThanhVu Nguyen Ugur Koc Javran Cheng Jeffrey S. Foster Adam A. Porter
University of Maryland, College Park
{tnguyen, ukoc, javran, jfoster, aporter}@cs.umd.edu

ABSTRACT

To develop, analyze, and evolve today's highly configurable software systems, developers need deep knowledge of a system's configuration options, e.g., how options need to be set to reach certain locations, what configurations to use for testing, etc. Today, acquiring this detailed information requires manual effort that is difficult, expensive, and error prone. In this paper, we propose iGen, a novel, lightweight dynamic analysis technique that automatically discovers a program's *interactions*—expressive logical formulae that give developers rich and detailed information about how a system's configuration option settings map to particular code coverage. iGen employs an iterative algorithm that runs a system under a small set of configurations, capturing coverage data; processes the coverage data to infer potential interactions; and then generates new configurations to further refine interactions in the next iteration. We evaluated iGen on 29 programs spanning five languages; the breadth of this study would be unachievable using prior interaction inference tools. Our results show that iGen finds precise interactions based on a very small fraction of the number of possible configurations. Moreover, iGen's results confirm several earlier hypotheses about typical interaction distributions and structures.

CCS Concepts

•Software and its engineering → Software testing and debugging; Software configuration management and version control systems; Dynamic analysis;

Keywords

Program analysis; software testing; configurable systems; dynamic analysis

1. INTRODUCTION

Modern software systems are increasingly designed to be configurable. This has many benefits, but it also greatly complicates tasks such as testing, debugging, and impact analysis because the total number of configurations can be very large. To carry out such tasks, developers must take advantage of task-specific structure in a system's configuration space. For example, a developer may observe that many configurations are the same in terms of the coverage they achieve under a test suite, and thus developers can perform effective testing using just a small set of configurations.

In prior work [22, 26, 27], we showed how to automatically infer *interactions* that concisely describe a system's configurations. Our focus is coverage, and we formally define an *interaction* to be a formula ϕ over configuration options such that (a) any configuration satisfying ϕ covers some location L under a given test suite and (b) ϕ is the logically weakest such formula (i.e., if ψ also describes configurations covering L then $\psi \Rightarrow \phi$). Thus, by knowing a system's interactions, a developer can determine useful information about configurations, e.g., given a location, determine what configurations cover it; given an interaction, determine what locations it covers; find important options and compute a minimal set of configurations to achieve certain coverage; etc. In the literature, feature interactions and presence conditions (Section 5) are similar to interactions and can explain functional (e.g., bug triggers, memory leaks) and non-functional (e.g., performance anomalies, power consumption) behaviors. Interactions can also aid reverse engineering and impact analysis [2, 24].

While our prior work was promising, it has significant limitations. In our first effort, we inferred interactions using Otter, a symbolic executor for C [22]. However, symbolic execution does not scale to large systems, even when restricted to configuration options; is brittle in the presence of frameworks, libraries, and native code; and is language-specific. In our second effort, we developed iTree [27], which infers interactions using dynamic analysis and machine learning. However, iTree's focus is on finding configurations to maximize coverage, and in practice it only discovers a small set of interactions. (Section 5 discusses these prior systems in more detail.)

In this paper, we introduce iGen, a new dynamic analysis tool for automatically discovering a program's interactions. iGen works by iteratively running a subject program under a test suite and set of configurations; inferring potential interactions from the resulting coverage information; and then generating new configurations that aim to refine the inferred interactions on the next iteration. By carefully choosing new configurations in this last step, iGen is able to quickly converge to a final, precise set of interactions using

only a small set of configurations.

Moreover, iGen's design overcomes the limitations of Otter and iTree. The only language-specific portion of iGen is obtaining code coverage, which is widely available for almost any language. iGen's analysis is also very lightweight and scalable, because, as we explain later, it uses simple computations over coverage information to infer interactions. Finally, although we did not mention it earlier, our prior work was restricted to inferring interactions that are purely conjunctive. In contrast, iGen supports interactions that are purely conjunctive, purely disjunctive, and specific mixtures of the two. (Section 2 describes iGen in more detail.)

We evaluated iGen by running it on 29 programs, far more than we were ever able to use with Otter or iTree. Our subject programs span five languages (C, Perl, Python, Haskell, and OCaml); range in size from tens of lines to hundreds of thousands of lines; and have between 2 and 50 configuration options. For most programs we apply iGen to run-time configuration options, but for one program, `httpd`, we study compile-time configuration options. (Section 3 describes our subject programs.)

We considered three research questions. First, we evaluated the correctness of iGen's inferred interactions. We found that, for a subset of the subject programs, the interactions produced by iGen's iterative algorithm are largely similar to what iGen would produce if it inferred interactions from *all* possible configurations. This suggests iGen does converge to the optimal solution. We also manually inspected a subset of the interactions found by iGen and verified they match the logic in the code.

Second, we measured iGen's performance and found it explores a very small fraction of the number of possible configurations. Moreover, like the work of Reisner et al., iGen generates dramatically more precise interactions than iTree. Yet, it runs in a small fraction of the time required for Reisner et al.'s experiments.

Finally, we analyzed iGen's output to learn interesting properties about the subject programs. We confirmed several results found in prior work [22, 26, 27], among them: the number of interactions is far smaller than what is combinatorially possible; yet a few (very) long interactions are needed for full coverage; and *enabling options*, which must be set a certain way to achieve most coverage, are common. We should emphasize that our prior work hypothesized these based cumulatively on just four programs in one language, whereas we observe them based on 29 programs in five languages. We also observed new phenomena: that disjunctive and mixed interactions are less common than conjunctive ones, but nonetheless cover a non-trivial number of lines. Finally, we showed that iGen's interactions can be used to compute a small set of configurations that cover all or most lines of the programs. (Section 4 reports on our evaluation.)

We believe iGen takes an important step forward in the practical understanding of configurable systems.

2. IGEN ALGORITHM

We begin our presentation by describing the iGen algorithm, whose pseudo-code is shown in Figure 1. The input to iGen is a program P and a test suite T, and the output is a set of interactions for locations in P that were covered when running on T. iGen works by iteratively generating a set of configurations (`configs` in the algorithm) until the coverage (`cov`) and interactions (`ints`) inferred from that set

```
input  : a program P and a test suite T
output : a set of interactions of P
1  cov, ints ← ∅
2  configs ← oneWayCoveringArray() ∪ {default_config}
3  while true do
4      old_cov ← cov
5      old_ints ← ints
6      cov, ncov ← runTestSuite(P, T, configs)
7      // returns cov(l) = {c | c covers l}
8      //         ncov(l) = {c | c does not cover l}
9      foreach location l ∈ cov do
10         conj ← ⊔ cov(l)
11         disj ← ¬(⊔ ncov(l))
12         disj' ← ¬ ⊔ {c | c ∈ ncov(l) ∧ c ⇒ conj}
13         conjdisj ← conj ∧ disj'
14         conj' ← ⊔{c | c ∈ cov(l) ∧ disj ⇒ ¬c}
15         disjconj ← disj ∨ conj'
16         ints(l) ← (conj, disj, conjdisj, disjconj)
17     if cov = old_cov ∧ ints = old_ints then break
18     configs ← genNewConfigs(ints)
19 foreach location l ∈ cov do
20     ints(l) ← check(ints(l), cov(l))
21     result(l) ← selStrongest(ints(l))
22 return result
```

Figure 1: iGen's iterative algorithm for inferring program interactions.

reach a fix-point.

The algorithm begins on line 2 by initializing `configs` to a randomly generated 1-way covering array [5, 6], i.e., it contains all possible settings of each individual option. The algorithm also includes a default configuration if one is available. In our experience, such a configuration typically yields high coverage under the test suite and hence is a useful starting point. On line 6, iGen runs the test suite under the current set of configurations,[1] producing two coverage maps: `cov` maps each location l to the set of configurations c such that at least one test covers l under c, and `ncov` maps l to the set of configurations that do not cover l.

Then for each location l covered by the test suite under some configuration (line 9), iGen infers candidate interactions. Although in theory interactions can be arbitrary formulae, iGen keeps its inference process efficient by assuming interactions follow particular syntactic *templates*. As it iterates, iGen computes the most precise interaction for each location for each template. At the end of the algorithm, iGen selects the *strongest* (in a logical sense) interaction per location across the different templates.

Currently, iGen supports four templates: `conj`, a purely conjunctive interaction; `disj`, a purely disjunctive interaction; `conjdisj`, a conjunctive interaction where the last conjunct is a disjunct; and `disjconj`, a disjunctive interaction where the last disjunct is a conjunct. We explain the computation of the interactions in detail below. This particular set

[1] In practice iGen memoizes the coverage information from previous runs and only runs the test suite under the new configurations.

```
// options: s, t, u, v, x, y, z
int max_z = 3;

if (x && y) {
  printf ("L0\n"); // x ∧ y
  if (!(0 < z && z < max_z)){
    printf ("L1\n"); // x ∧ y ∧ (z ∈ {0, 3, 4})
  }
}else{
  printf ("L2\n"); // ¬x ∨ ¬y
}
printf ("L3\n"); // true
if (u && v) {
  printf ("L4\n"); // u ∧ v
  if (s || t) {
    printf ("L5\n"); // u ∧ v ∧ (s ∨ t)
  }
}
```

Figure 2: Program with seven configuration options. Locations L0–L5 are annotated with associated interactions.

of templates was chosen partially based on our experience (e.g., we believe conjunctive interactions are very common) and partially based on what is efficient to compute (e.g., mixing one disjunction into a conjunction or vice-versa is a relatively small cost, whereas more complex interleavings of conjunctions and disjunctions would be much less efficient.)

After saving candidate interactions (line 16), the loop terminates if iGen has reached a fix-point (line 17). Otherwise, iGen creates additional configurations (line 18) designed to refine interactions (details below) and continues iteration.

After the main loop terminates, there are two steps remaining. First, because of some heuristics in iGen's interaction generation, some interactions it computes may not actually cover the expected lines. Thus on line 20, iGen iterates through the set of interactions and checks that for any interaction ϕ for l, it is actually the case that $c \Rightarrow \phi$ for all configurations c that cover l. iGen eliminates any interaction that fails this check by setting it to *true*. Second, on line 21, iGen and sets result(l) to be the logically strongest interaction among conj, disj, conjdisjm, and disjconj. If there is no single strongest interaction, iGen eliminates any interactions that are weaker than another and returns the conjunction of the remaining strongest interactions.

Running Example. We next use the C program in Figure 2 to explain the details of iGen. This program has seven configuration options, listed on the first line of the figure. The first six options are boolean-valued, and the last one, z, ranges over the set $\{0, 1, 2, 3, 4\}$. Thus, this program has $2^6 \times 5 = 320$ possible configurations.

The code in Figure 2 includes print statements that mark six locations $L0$–$L5$. At each location, we list the associated desired interaction. For example, $L1$ is covered by any configuration in which x and y are true and z is 0, 3, or 4. As another example, $L3$ is covered by any configuration, hence its interaction is *true*.

Prior approaches to interaction inference are not sufficient for this example. The work of Reisner et. al [22] only supports conjunctions, so it must approximate the interactions for $L1$, $L2$, and $L5$. iTree [27] actually produces no interactions for this example, because all lines are covered by iTree's initial two-way covering array (iTree stops generating interactions when no new coverage is achieved).

For this example, iGen initializes configs to the following covering array (there is no default configuration):

config	s	t	u	v	x	y	z	coverage
c_1	0	0	1	1	1	0	1	$L2, L3, L4$
c_2	1	1	0	0	1	1	0	$L0, L1, L3$
c_3	0	0	1	1	0	0	2	$L2, L3, L4$
c_4	0	0	1	1	1	1	3	$L0, L1, L3, L4$
c_5	0	1	1	1	1	0	4	$L2, L3, L4, L5$

We list the coverage of each configuration on the right.

Conjunctive Interactions. The first interaction template, conj, supports conjunctions of *membership constraints* $x \in S$ indicating option x ranges over set S. For example, the interaction for $L1$ in Figure 2 is shorthand for $(x \in \{1\}) \wedge (y \in \{1\}) \wedge (z \in \{0, 3, 4\})$. On line 10 of the algorithm, iGen infers conj by taking the *pointwise union* of the covering configurations' option settings and then conjoining them. We denote this operation by $\sqcup$. For example, the table below shows the pointwise union of the two covering configurations c_2 and c_4. Here $\top$ is the universal set for an option.

$L1$	s	t	u	v	x	y	z
c_2	1	1	0	0	1	1	0
c_4	0	0	1	1	1	1	3
union	$\top$	$\top$	$\top$	$\top$	1	1	0, 3

Thus to form conj $= c_2 \sqcup c_4$ for $L1$ we simply conjoin the option settings from the above table to yield conj $= x \wedge y \wedge (z \in \{0, 3\})$, where we write x and y for $x \in \{1\}$ and $y \in \{1\}$. Note we omit constraints corresponding to $\top$, since those indicate options that can take any value.

At this point, conj is close to, but not quite, the correct interaction for $L1$. The problem is that configs is missing a configuration where $x = y = 1$ and $z = 4$. Thus—skipping over the other templates and other locations for the moment—for the next iteration iGen generates additional configurations to *refine* the set of interactions, using the genNewConfigs call on line 18.

iGen derives these new configurations by systematically changing the settings from one selected interaction from ints. For example, genNewConfigs might generate new configurations from interaction conj to yield:

config	s	t	u	v	x	y	z	coverage
c_6	1	0	1	0	0	1	0	$L2, L3$
c_7	0	0	0	1	1	0	3	$L2, L3$
c_8	1	1	0	1	1	1	1	$L0, L3$
c_9	1	0	1	0	1	1	2	$L0, L3$
c_{10}	1	0	0	1	1	1	4	$L0, L1, L3$

Here each configuration disagrees with conj $= x \wedge y \wedge (z \in \{0, 3\})$ in one setting, e.g., c_6 has $\neg x$, c_7 has $\neg y$, and c_8 has $z = 1$. Then the next iteration of the fix-point loop will compute conj for $L1$ from c_2, c_4, and c_{10}. Since c_{10} has $x = y = 1$ and $z = 4$ (which was not covered in the first set of configurations), iGen produces the correct interaction $x \wedge y \wedge (z \in \{0, 3, 4\})$.

In practice, we could choose any interaction for any line and use it to generate new configurations. Currently, iGen's

heuristic is to choose the *longest* current interaction, based on our prior experience suggesting long interactions are uncommon and hence likely to be inaccurate. If there is a tie for longest interaction, iGen selects randomly among the longest. If iGen selects an interaction that does not fully constrain some configuration options, then it assigns random values (satisfying whatever constraint in present) to those options when creating new configurations.

Disjunctive Interactions. Next let us consider the interaction $\neg x \vee \neg y$ for $L2$ in Figure 2. By construction, conj cannot encode this formula—although the membership constraints in conj are a form of disjunction (e.g., $z \in \{0,3\}$ is the same as $z = 0 \vee z = 3$), they cannot represent disjunctions among different variables.

There are a variety of potential ways to infer more general disjunctions, but we want to maintain the same efficiency as inferring conjunctive interactions. To motivate iGen's approach to disjunctions, observe that $L2$'s interaction arises because an else branch was taken. In fact, $L2$'s interaction is exactly the negation of the interaction for $L0$ from the true branch. Thus, iGen computes disjunctive interactions by first computing a non-covering interaction, which is a *conjunctive* interaction for the configurations that do *not* cover line $L2$, and then *negates* it to get a disjunctive interaction for $L2$ (line 11). In our running example, c_2 and c_4 are the only configurations that do not cover $L2$, thus iGen computes $c_2 \sqcup c_4 = x \wedge y \wedge (z \in \{0,3\})$. Negating that yields $\mathsf{disj} = \neg x \vee \neg y \ \vee \ (z \in \{1,2,4\})$, which is close to the correct interaction for $L2$.

Notice this approach to disjunctions is a straightforward extension of conjunctive interaction inference. Also notice that it is heuristic since the computed interaction may not actually cover the given line; thus disjunctive interactions may be eliminated on line 20 of the algorithm in Figure 1.

Disjunctive interactions can be refined in two ways. First, they may be refined by coincidence if genNewConfigs selects a long conjunctive interaction to refine. Second, genNewConfigs also considers the negation of disj as a possible longest interaction to use for refinement (essentially refining an interaction describing configurations that do not reach the current location).

Mixed Interactions. Finally, some interactions require mixtures of conjunctions and disjunctions, such as the interaction $u \wedge v \wedge (s \vee t)$ for $L5$. Looking at Figure 2, notice this interaction occurs because a disjunctive condition is nested inside of a conjunctive condition—in fact, the interaction for $L5$ is the interaction for $L4$ with one additional clause.

This motivates iGen's approach to inferring mixed interactions by *extending* shorter interactions. Lines 12–13 give the code for computing conjdisj. Recall that to compute the pure disjunction disj, iGen negates the pointwise union of non-covering configurations. On line 12 we use the same idea to compute disj', but instead of *all* non-covering configurations, we only include the non-covering configurations that satisfy conj. Essentially we are projecting the iGen algorithm onto just configurations that satisfy that interaction. Thus, when we infer the disjunction disj', we conjoin it onto conj to compute the final mixed interaction.

For our running example, after several iterations conj for $L5$ will be $u \wedge v$ (details not shown). Out of the configurations that do not cover $L5$, only c_1, c_3, and c_4 also satisfy $u \wedge v$. Thus disj' will be $\neg(c_1 \sqcup c_3 \sqcup c_4) = s \vee \ t \vee \neg u \vee \neg v \vee (z \in \{0,4\})$. Thus after some simplification we get $\mathsf{conj} \wedge \mathsf{disj}' = u \wedge v \wedge (s \vee t \vee (z \in \{0,4\}))$, which is almost the interaction for $L5$. After further iteration, iGen eventually reaches the final, fully precise interaction for $L5$.

Using the dual of the above approach, lines 14–15 infer another mixed interaction disjconj by extending the computed disjunctive interaction disj with a conjunction conj'. Here conj' is generated just like conj, but we only include configurations that disagree in some setting with some clause of disj, since otherwise the configuration is already included in the left side of the disjunct on line 15.

Notice that iGen's approach for inferring mixed interactions maintains the efficiency of computing pure conjunctions and disjunctions. We could extend the algorithm further to compute conjunctions with nested disjunctions with nested conjunctions etc., but we have not explored that yet.

Lastly, in addition to considering conj and disj as potential longest interactions, genNewConfigs also considers conj'; and (negated) disj'. Thus, mixed interactions may be refined whenever one of those four components is refined.

Discussion. Putting this all together, after running to completion, iGen produces the same interactions for our example as in the comments in Figure 2. Moreover, iGen finds these interactions by analyzing just 37 configurations instead of 320 possible configurations. The experiments in Section 4 show that iGen analyzes an even smaller fraction of the possible configurations on programs with a large number of configuration options.

As mentioned above, iGen has several sources of randomness: the one-way covering array, the interaction used for generating new configurations, and the values of un- or under-constrained option settings in those new configurations. Thus, iGen is actually a stochastic algorithm that may produce slightly different results each time. However, in our experiments we demonstrate that the variance is reasonable.

Moreover, the computation of each iteration of iGen is straightforward and efficient. Pointwise union is linear in the number of configurations and options. Checking the various implications is done with an SMT solver, which is very efficient in practice. As discussed in Section 4.2, iGen's running time is mostly consumed by running the test suite (line 6).

Finally, notice that if iGen were to iterate until it had generated *all* configurations, then it would be guaranteed to produce correct interactions if they fall under the given templates. For example, suppose some location L has a purely conjunctive interaction $x \wedge y$. Then if we consider the set of *all* configurations that cover L, they all satisfy $x \wedge y$; but, the set has configurations that differ in every possible way for options that are not x and y. Thus, pointwise union of this set will yield x and y as *true* and every other option as $\top$. Hence iGen must produce the correct interaction $x \wedge y$. Similar arguments follow for the other interaction templates. In Section 4.1, we take advantage of this observation to help evaluate the correctness of iGen's inferred interactions.

3. SUBJECT PROGRAMS

iGen is implemented in approximately 2,500 lines of Python. It uses the Z3 SMT solver [9] to reason about implications. We computed line coverage using gcov for C, python-cov [21] for Python, and MDevel::Cover [16] for Perl. We computed

Table 1: Subject programs.

prog	lang	ver	loc	opts	cspace	tests
id	C	8.23	332	10	1024	4
uname	C	8.23	281	11	2048	2
cat	C	8.23	496	12	4096	12
mv	C	8.23	375	11	5120	14
ln	C	8.23	478	12	10 240	14
date	C	8.23	469	7	17 280	11
join	C	8.23	892	12	18 432	8
sort	C	8.23	3348	22	6 291 456	9
ls	C	8.23	3545	47	3.5×10^{14}	16
p-id	Perl	0.14	131	8	256	4
p-uname	Perl	0.14	25	6	64	2
p-cat	Perl	0.14	47	7	128	12
p-ln	Perl	0.14	62	2	4	14
p-date	Perl	0.14	136	5	3360	11
p-join	Perl	0.14	178	10	4608	8
p-sort	Perl	0.14	399	11	2048	9
p-ls	Perl	0.14	403	26	6.7×10^{7}	16
cloc	Perl	1.62	8014	19	524 288	296
ack	Perl	2.14	2711	32	4.3×10^{9}	5
grin	Python	1.2.1	628	21	2 097 152	5
pylint	Python	1.3.1	7837	29	5.8×10^{10}	93
hlint	Haskell	1.9.21	3266	12	8192	594
pandoc	Haskell	1.13.2	24 755	22	4.0×10^{9}	42
unison	OCaml	2.48.3	29 796	16	393 216	5
bibtex2html	OCaml	1.98	9172	33	1.2×10^{9}	3
gzip	C	1.6	32 080	17	131 072	10
httpd	C	2.2.29	238 345	50	1.1×10^{15}	400
vsftpd	C	2.0.7	10 482	30	2.1×10^{9}	64
ngircd	C	0.12.0	13 601	13	29 764	141

expression coverage using Bisect [3] for OCaml and Hpc [14] for Haskell.

Our experiments were performed on a 2.40GHz Intel Xeon CPU with 16 GB RAM running RedHat Enterprise Linux 5.11 (64-bit). The source code for iGen is available at https://bitbucket.org/nguyenthanhvuh/igen.

Programs. Table 1 lists our subject programs. For each program, we list its name, language, version, and lines of code as measured by SLOCCount [25]. Note that the line count is typically higher than the number of locations reachable by the test suite. We also report the number of configuration options (opts) and the total number of possible configurations (cspace). Finally, we list the number of test cases in the program's test suite.

The first group of programs comes from the widely used GNU coreutils. These programs are configured via command-line options. We selected a subset of coreutils with relatively large configuration spaces (at least 1024 configurations each). The second group comprises coreutils reimplemented in the Perl Power Tools (PPT) project [20] (excluding mv which was not implemented in PPT). These programs are named as the coreutils programs but with a prefix of p-.

The third group contains an assortment of programs to demonstrate iGen's wide applicability. Briefly: cloc is a lines-of-code counter; ack and grin are grep-like programs; pylint and hlint are static checkers for Python and Haskell, respectively; pandoc is a document converter; unison is a file synchronizer; bibtex2html converts BibTeX files to HTML; gzip is a compression tool; and finally, httpd is the well-known Apache http server. Cumulatively these programs span five languages (two programs per languages) and range from a few thousand to hundreds of thousands of lines.

The last group comprises vsftpd, a highly secure ftp server, and ngircd, an IRC daemon. These programs were also studied by Reisner et al. [22], who used Otter, a symbolic execution tool, to exhaustively compute all possible program executions under all possible settings of certain configuration options. To make a direct comparison to Reisner et al.'s work possible, we ran iGen on these programs in a special mode in which, rather than running a test suite, we used Otter's output as an oracle of which lines are reachable under which configurations.

Configuration Options. We selected configuration options for study in a variety of ways. We studied all options for coreutils. Most of these options are boolean-valued, but nine can take on a wider but finite range of values, all of which we included, e.g., we include all possible formats date accepts. We omit options that range over an unbounded set of values. For PPT, we studied the same options as coreutils when available, though PPT only supports a small subset of coreutils' options.

For the programs in the third group, we used the run-time options—for httpd, the compile time options—we could get working correctly. For example, we excluded httpd options that caused compiler errors when we changed them. We ignore options that can take arbitrary values, e.g., pylint options that take a regexp or Python expression as input.

Most of the options we selected are boolean-valued, but several range over a finite set of values, e.g., we consider seven highlight options for formatting in pandoc and three permission modes for modifying files in unison.

We used the same options for vsftpd and ngircd as Reisner et al., to make a direct comparison possible.

Test Suites. We manually created tests for coreutils that cover common command usage. For example, for cat, we wrote tests that read a text file, a binary file, a non-existent file, results piped from other commands, etc. We used the same tests for coreutils and PPT.

For the third group of programs, test selection varied. For httpd, hlint, pylint, and bibtex2html, we used the default tests. For the remaining programs, we started from the default tests (which were relatively limited) and added more tests to cover basic functionality.

4. EVALUATION

We consider three research questions:

- **R1 (correctness)**: Does iGen generate correct interactions?
- **R2 (efficiency)**: What are iGen's performance characteristics?
- **R3 (analysis)**: What can we learn from inferred interactions?

To investigate these questions, we applied iGen to the subject programs described in Section 3. Table 2 summarizes the results and reports the medians across 21 runs and their variance[2] as the semi-interquartile range (SIQR). For each

[2]Recall from Section 2 that iGen uses randomness, so different runs may produce slightly different results.

Table 2: iGen's results for the benchmark programs shown in Table 1. Numbers in regular font are medians across 21 runs. Numbers in small font are semi-interquartile ranges measuring variance among the runs.

					time (s)				interactions							
prog	configs		cov		search		total		conj		disj		mix		total	
id	157	5	138	0	18	1	34	3	23	0	1	0	1	0	25	0
uname	95	5	87	0	9	1	15	1	16	0	2	0	7	1	25	1
cat	131	6	204	0	15	1	42	5	18	0	1	0	6	0	25	0
mv	106	9	172	0	9	1	38	2	16	0	1	0	1	0	18	0
ln	213	18	162	0	32	4	96	13	20	0	1	0	5	0	26	0
date	680	44	127	0	97	15	350	94	11	0	1	0	2	0	14	0
join	323	21	382	0	77	9	158	25	28	0	6	0	8	1	32	1
sort	1346	68	1083	0	2003	322	3113	379	78	1	2	0	13	1	93	2
ls	2175	250	1034	0	5091	823	9837	1887	109	0	2	1	8	0	120	0
p-id	82	2	73	0	5	0	283	7	9	0	0	0	0	0	9	0
p-uname	28	0	19	0	1	0	62	1	1	0	5	0	0	0	6	0
p-cat	26	1	30	0	1	0	246	11	3	0	1	0	0	0	4	0
p-ln	4	0	36	0	0	0	42	0	3	0	0	0	0	0	3	0
p-date	160	0	40	0	5	0	2061	159	5	0	0	0	0	0	5	0
p-join	111	21	114	0	8	2	1573	267	12	0	3	0	2	1	17	1
p-sort	116	5	191	0	11	1	3947	167	18	0	1	0	7	0	26	0
p-ls	272	11	216	0	52	2	13 803	842	25	0	1	0	3	0	29	0
cloc	210	9	972	0	67	5	5017	456	21	0	2	0	13	0	36	0
ack	1347	42	867	0	1962	88	23 127	999	53	1	1	0	5	0	59	1
grin	242	28	331	0	35	6	411	51	9	0	0	0	9	0	18	0
pylint	1916	143	5712	0	5637	536	27 175	2553	54	5	1	0	17	5	72	1
hlint	328	18	6757	0	4365	28	9525	761	22	0	1	0	12	0	35	0
pandoc	653	56	31 635	0	2284	451	23 515	2231	83	0	7	0	12	1	102	1
unison	381	27	3784	0	383	38	4641	341	36	0	1	0	13	1	50	1
bibtex2html	670	118	1345	0	369	90	667	136	90	0	0	0	15	0	105	0
gzip	495	27	1635	0	251	29	12 029	486	27	3	5	0	11	3	43	6
httpd	838	114	10 633	1	3596	777	197 390	30012	104	3	0	0	9	6	113	3
vsftpd	620	38	2549	0	628	125	652	126	42	0	3	0	4	0	49	0
ngircd	650	46	3090	0	820	88	1469	125	34	0	3	0	9	0	46	0

program, columns configs and cov show the number of configurations iGen created and the number of locations covered by these configurations, respectively. The next two columns show iGen's running time in seconds: total is the total time and search is the time excluding the time to run the test suite. The remaining columns list the number of inferred interactions, divided into conjunctive (conj), disjunctive (disj), and mixed interactions (mix), with the cumulative sum on the right (total). The low SIQR for inferred coverage and interactions indicate iGen produces relatively stable output across runs.

4.1 RQ1: Correctness

Exhaustive Runs. To measure the correctness of the inferred interactions, we first evaluated whether iGen produces the same results with its iterative algorithm as it could produce if it used all configurations. Recall from Section 2 that running iGen with all configurations is guaranteed to find correct interactions if they match our templates, so this essentially gives us ground truth with respect to the test suite.

To perform this evaluation, we selected the fourteen programs with the smallest configuration spaces and ran one loop of iGen with configs (Figure 1) initialized to the set of all configurations. We also used exhaustive symbolic execution information for vsftpd and ngircd [22] to simulate iGen running with all configurations of those programs.

Table 3: Comparing iGen to exhaustive runs.

prog	δ cov	f-score	prog	δ cov	f-score
id	0	0.98	p-id	0	0.97
uname	0	0.97	p-uname	0	1.00
cat	0	1.00	p-cat	0	1.00
mv	0	0.94	p-ln	0	1.00
ln	0	0.99	p-date	-2	0.35
date	0	0.94	p-join	0	0.77
join	-1	0.99	p-sort	0	1.00
vsftpd	0	0.997	ngircd	0	0.92

Table 3 shows these comparisons. The second column δ cov reports the differences between the number of lines covered by iGen's regular runs and those covered by the exhaustive runs. Overall we see iGen generates interactions with coverage very similar to the exhaustive runs. In total, iGen missed one line in join and two lines in p-date. We investigated and found these three uncovered lines are guarded by long conjunctive interactions. For example, iGen missed join.c:997, which is guarded by an interaction with 10 conjuncts.

Column f-score measures the accuracy of iGen's runs compared to the exhaustive runs using a *balanced f-score* [23],

which ranges between 0 and 1, with 1 representing perfect agreement. In more detail, the f-score is based on comparing for every location whether settings in the interaction for that location match between the standard and exhaustive runs. Notice this is a very strict test.

Table 3 shows that iGen generates mostly the same interactions for most programs as the exhaustive runs. We investigated the two outliers, p-date and p-join, and found the low f-scores are due to two factors. First, these programs have few interactions (5 for p-date and 17 for p-join), so a few differences cause a large score change. Second, iGen almost but does not quite infer the right interactions, which still results in an f-score penalty. For example, for p-date, instead of inferring the (correct) interaction $G \wedge d \wedge (\neg R \vee \neg iso8601)$, iGen infers the less precise interaction $G \wedge d$. Similar near-matches account for most of the f-score differences of the remaining programs, e.g., in ngircd, iGen generates several mixed interactions where it should generate conjunctive interactions.

Manual Inspection. We also manually analyzed the program source code to make sure certain interesting or non-obvious interactions generated by iGen are correct. We focused on coreutils because these programs are small enough for careful manual inspection and they allow for interesting comparisons to PPT.

For p-uname, iGen discovers an interaction a ∨ m covering the line uname.pl:27, which prints the machine name. This interaction thus specifies (correctly) that the machine name is printed when given either option a or m. iGen also discovers other similar interactions for p-uname, including a ∨ o, which prints the operating system name, a∨n to print the hostname, etc. For the corresponding uname command iGen found similar but longer interactions such as (¬help ∧ ¬verbose) ∧ (a ∨ m), which prints the machine name but only when help and verbose are not given (note these options are not supported by PPT). We refer to options like help as *enabling options* [22, 27], since they must be set a certain way to achieve significant coverage. Notice also that this is a mixed interaction, which cannot be inferred by our prior work.

Another interesting interaction iGen found is (¬A ∧ ¬t ∧ ¬T) ∧ (e ∨ v) covering line cat.c:462 of cat. Reviewing the source code, we found this line is only executed when the global variables showtab and shownonprinting are false and true, respectively. Further examination reveals that showtab is false by default but any of options A, t, or T cause it to be true; thus all of these options must not be given to cover the considered line. Moreover, shownonprinting is only true when either e, t, v, or A set; thus one of these options must be given to cover the considered line. Putting this logic together and simplifying yields the interaction iGen discovered.

We were surprised that for uname, iGen generates a purely conjunctive interaction containing the negation of all options. Inspection of the source code reveals this is correct—one line of code is only invoked when no options are given. Interestingly, p-uname does not have this interaction, and indeed does not behave as uname when run with no options.

Finally, we investigated the cases of non-zero SIQRs in Table 2 and found that most differences involve disjunctions. For example, 13 of 21 uname runs found the interaction a ∨ s ∨ n ∨ r ∨ v ∨ m ∨ p ∨ i ∨ o ∨ help ∨ version by taking a shorter conjunctive interaction, e.g., i ∧ ¬a ∧ ¬help ∧ ¬version; modifying its settings, yielding, among others, ¬i∧¬a∧¬help∧¬version; and randomly assigning remaining options to get ¬a∧¬s∧¬n∧¬r∧¬v∧¬m∧¬p∧¬i∧¬o∧¬help∧¬version, which is then negated. While iGen performs random assignment for many different short interactions, the chance of getting that exact conjunction are still low overall—and in 8 of 21 uname runs, iGen misses that interaction. In general, non-zero SIQRs arise from lower probability events like this.

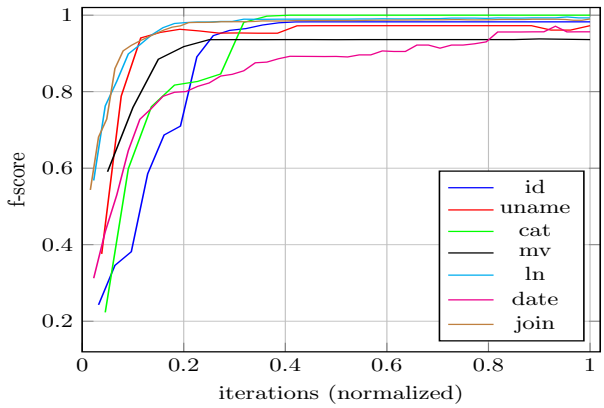

Figure 3: Progress of iGen on generating interactions for GNU coreutils programs.

4.2 RQ2: Performance

Table 2 shows that for most programs, iGen's running time is dominated by running the test suite. Thus, programs with larger configuration spaces tend to take longer because iGen has to create more configurations to analyze these programs. Nonetheless, comparing to Table 1, we see that iGen scales well to large programs because it only explores a small fraction of the total possible number of configurations. Interestingly, Table 2 shows the number of explored configurations is not directly proportional to the configuration space size. For example, ls has eight orders of magnitude more possible configurations than sort, but iGen only explores 1.5× more configurations.

We believe that iGen represents a good trade-off between correctness and efficiency compared to previous work. On the one hand, iGen generates more configurations (and hence is slower) than iTree. For vsftpd and ngircd, iGen generates 620 and 650 configurations, respectively, while iTree creates around 100 configurations each [27]. However, iGen generates much more precise interaction information—iTree reports only 4 interactions for vsftpd and 3 interactions for ngircd, compared to iGen's 49 and 46 interactions, respectively.

On the other hand, iGen runs dramatically faster than the experiments by Reisner et al. [22], which took two weeks to analyze just a few programs using a specialized cluster. Moreover, although iGen is not guaranteed to produce precise interactions, the results discussed in Section 4.1 suggest iGen is as precise as Reisner et al.'s work in practice.

Convergence. Figure 3 shows how iGen converges to its final results on a subset of coreutils. The x-axis is the iteration count (normalized such that 1 represents all iterations for that particular program), and the y-axis is the f-score compared to the exhaustive runs. These results show that iGen converges fairly quickly. After approximately 20% of

Table 4: Comparing random search to exhaustive runs.

prog	δ cov	f-score	prog	δ cov	f-score
id	-3	0.85	p-id	-2	0.88
uname	-1	0.83	p-uname	0	0.94
cat	-1	0.93	p-cat	-1	0.93
mv	0	0.58	p-ln	0	1.00
ln	-4	0.76	p-date	-2	0.33
date	-4	0.51	p-join	-6	0.70
join	-17	0.82	p-sort	-2	0.90
vsftpd	-356	0.58	ngircd	-1289	0.27

the iterations, iGen's f-score has reached 0.8 or more. This suggests an iGen user could potential cut off iteration early and still achieve reasonable results.

iGen's convergence relies critically on genNewConfigs (Figure 1 line 18) generating useful configurations. Table 4 demonstrates this by showing iGen runs on randomly selected configurations. More specifically, for each program in Table 3, we generated the same number of random configurations as the number of configurations iGen used (Table 2). We also include the default configuration, if it exists. We next ran the iGen main loop once on these configurations to compute the results, and then compared coverage information and f-score to the exhaustive runs.

Comparing to Table 3, we see iGen generates much more precise interactions and has better coverage than random search. The differences are most significant for larger programs, e.g., the most extreme case is ngircd, where random search has f-score 0.27 and covers 1289 fewer lines than exhaustive run, while iGen has f-score 0.92 and has the same coverage as the exhaustive run.

4.3 RQ3: Analysis

We analyzed iGen's results in detail to learn interesting properties of the subject programs. In the following, an interaction's *length* is the number of options it contains.[3]

Disjunctive and Mixed Interactions. We observe that many programs require non-conjunctive interactions. Table 2 shows that approximately 20% of the inferred interactions are disjunctions or mixed interactions. Furthermore, the analysis in Section 4.1 shows that mixed interactions, though rare, do exist in real-world software. Thus, these interactions, which were omitted from prior work [22, 26, 27], are important.

Interaction Length and Coverage. In prior work we observed that the total number of interactions found is far fewer than the number of possible interactions [26, 27]. We observe the same trend in Table 2. For example, p-cat has 7 binary options, yielding 128 possible configurations and at least 4373 possible interactions. However, iGen finds four interactions, which is less than 0.1% of 4373. The same trend can be seen throughout the table.

We also observed in prior work that longer conjunctive interactions tend to contain shorter conjunctive interactions, e.g., if $a \wedge b \wedge c \wedge d \wedge e$ is an interaction, it is likely that a shorter formula like $a \wedge b$ is an interaction [22]. We manually examined iGen's interactions and found that this pattern also holds. For most programs, conjunctive interactions of length at least three include a shorter interaction. This is likely due to nested guards, e.g., such as the interaction on $L5$ of Figure 2, but with a non-disjunctive inner condition.

Table 5 looks at the interactions in more detail by showing the number of interactions iGen infers and the covered lines at each interaction length. The last column (max) reports the length of the longest interaction. We observe that the maximum interaction length for most programs is significantly smaller than the number of configuration options. However, there are five programs—id, uname, cat, p-join, and httpd—that have interactions that include all options. (We discussed the uname case in Section 4.1.)

Although some non-trivial coverage is achieved by large interactions (e.g., conjunctions of all options), 87% of the coverage is obtained by interactions of length less than three. These observations are similar to our prior work [22, 26, 27].

One exceptional case is ack, in which most coverage is achieved by large interactions. Investigating further, we found ack has many options that must be disabled for most of its functionality to be exercised.

Enabling Options. In addition to the enabling option help mentioned for uname, and the enabling options just mentioned for ack, we found many other examples of enabling options, including the following. For id, option Z must be disabled for most coverage (because that option is only applicable to a secured Linux kernel). For httpd, option –enable-http must be enabled for almost all coverage, and –enable-so (which allows for shared modules) must be enabled for a majority of the coverage. For vsftpd, disabling ssl and local, and enabling anon, are important to coverage. Finally, for ngircd, options ListenIPv4 and Conf_PredefChannelsOnly are important to coverage.

Notice these results depend on the test suite and environment. For example, an enabled option Z for id could be useful on a secured kernel, and ssl for vsftpd can be enabled when running with SSL certificates. This shows how iGen naturally adapts the inferred interactions to the current setting.

Minimal Covering Configurations. Finally, we used inferred interactions to compute a minimal set of configurations that achieve the full coverage found by iGen. To do so, we used the following greedy algorithm. Given a set of interactions, we first remove any interactions implied by others. For example, if $x \wedge y$ and x are interactions, we remove x, because a configuration satisfying $x \wedge y$ will also satisfy x. Next, we randomly conjoin interactions whose conjunction is satisfiable, e.g., we combine $x \wedge y$ and $z \vee w$ to yield $x \wedge y \wedge (z \vee w)$. This operation is greedy because it tries to combine as many compatible interactions as possible. Finally, we generate configurations that are compatible with the combined interactions, thus producing a small set of configurations that covers the same locations as the interactions. Note that this algorithm may not produce an optimal result, but in practice it is effective.

Table 6 summarizes the results. For each program we list the full configuration space (from Table 1) and the size of the minimal configuration set computed with our algorithm (again the coverage of these sets are similar as those as shown

[3]It is more typical to refer to this as the *strength* of an interaction. However, in our setting, longer conjunctive formulae are logically stronger than shorter conjunctive formulae, but longer disjunctive formulae are logically weaker than shorter disjunctive formulae. Hence we use *length* to avoid confusion.

Table 5: Number of interactions and covered lines per interaction length. Results are medians of 21 runs.

prog	0		1		2		3		4		5		6		7		8		9		10+		max
id	1	15	2	3	7	29	2	29	2	10	1	1	5	14	1	2	1	1	2	33	1	1	10
uname	1	10	10	32	2	32	-	-	9	11	-	-	-	-	-	-	-	-	-	-	2	2	11
cat	1	16	11	42	2	35	-	-	2	13	1	1	1	22	2	3	1	1	-	-	4	71	12
mv	1	51	9	36	3	53	3	11	1	1	1	21	-	-	-	-	-	-	-	-	-	-	5
ln	1	12	10	44	3	38	1	1	4	54	3	7	2	3	2	3	-	-	-	-	-	-	7
date	1	14	3	9	3	89	3	9	-	-	-	-	4	6	-	-	-	-	-	-	-	-	6
join	1	21	10	66	7	197	10	51	6	28	5	10	1	6	1	3	-	-	-	-	-	-	7
sort	1	82	11	25	4	183	18	70	6	82	7	344	13	92	4	48	3	11	2	3	21	87	15
ls	1	51	46	160	3	187	11	317	22	164	13	62	9	25	5	29	2	8	1	-	13	26	47
p-id	1	7	1	11	-	-	-	-	-	-	5	53	2	2	-	-	-	-	-	-	-	-	6
p-uname	1	14	-	-	5	5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2
p-cat	1	23	2	4	-	-	1	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3
p-ln	1	34	2	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1
p-date	1	8	4	32	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1
p-join	1	59	10	32	2	6	-	-	-	-	1	7	-	-	-	-	-	-	-	-	1	4	10
p-sort	1	42	6	43	6	34	2	8	3	49	5	9	2	4	1	2	-	-	-	-	-	-	7
p-ls	1	124	7	49	9	25	2	4	3	5	5	7	2	2	-	-	-	-	-	-	-	-	6
cloc	1	190	4	567	11	124	7	50	8	29	1	1	4	11	-	-	-	-	-	-	-	-	6
ack	1	105	1	2	2	7	2	17	2	14	2	57	1	36	2	48	4	52	6	57	34	464	15
grin	1	105	2	73	5	112	0	0	3	11	3	23	3	5	1	2	-	-	-	-	-	-	7
pylint	1	2062	1	4	3	44	5	39	2	101	6	348	15	1602	18	1436	13	62	6	12	-	-	9
hlint	1	288	1	13	2	2270	4	150	8	324	10	2978	3	61	3	8	2	641	1	24	-	-	9
pandoc	1	27 788	60	2674	25	1065	10	86	2	9	1	2	1	1	-	-	-	-	1	1	-	-	5
unison	1	983	3	1970	16	579	12	137	14	102	3	10	1	1	-	-	-	-	-	-	-	-	6
bibtex2html	1	372	35	419	6	149	19	143	16	150	22	90	5	19	-	-	-	-	-	-	-	-	6
gzip	1	103	2	9	4	182	6	370	5	206	4	86	4	12	3	19	2	28	1	1	7	47	17
httpd	1	104	1	708	39	5524	46	3765	13	370	4	40	3	90	1	4	-	-	-	-	50	1	50
vsftpd	1	336	4	101	6	170	4	1373	18	410	8	114	6	35	2	10	-	-	-	-	-	-	7
ngircd	1	748	3	460	4	525	16	827	14	457	6	68	1	2	-	-	-	-	-	-	-	-	6

Table 6: Using iGen's interactions to compute minimal covering configurations.

prog	cspace	min cspace
id	1024	10
uname	2048	4
cat	4096	6
mv	5120	4
ln	10 240	7
date	17 280	7
join	18 432	7
sort	6 291 456	17
ls	3.5×10^{14}	15
p-id	256	7
p-uname	64	1
p-cat	128	1
p-ln	4	1
p-date	3360	3
p-join	4608	4
p-sort	2048	6
p-ls	6.7×10^{7}	10
cloc	524 288	6
ack	4.3×10^{9}	13
grin	2 097 152	6
pylint	5.8×10^{10}	11
hlint	8192	5
pandoc	4.0×10^{9}	5
unison	393 216	7
bibtex2html	1.2×10^{9}	8
gzip	131 072	10
httpd	1.1×10^{15}	5
vsftpd	2.1×10^{9}	6
ngircd	29 764	6

in Table 2). Our results show that the minimal configuration set is dramatically smaller than the full configuration space. In prior work, we found similar results: We constructed a minimal line covering set of 5 configurations for vsftpd and 7 for ngricd [22]. We believe the small differences between those results and Table 6 are due to iGen supporting richer interactions and randomness in the greedy algorithm.

4.4 Threats to Validity

Like any empirical study, our conclusions are limited by potential threats to validity. As a result our findings may not generalize in certain ways or to certain systems.

In this work we examined several subject systems, covering a range of different sizes, from 25 to 238k lines of code, written in five different languages. Although each of these systems is realistic and widely used, the whole set of systems represents only a sample of all possible software systems. In addition, we focused on subsets of configuration options; the number of options was substantial, but we did not include every possible option, as discussed earlier.

Another potential threat is that iGen relies on running test suites to draw its conclusions. The test sets we used have reasonable, but not complete, coverage. Individually, the test cases tend to focus 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. Systems whose test suites are less (or more) complete could have different results.

Finally, iGen misses interactions that are not in one of the forms discussed in Section 2. However, from Table 2 we see that interactions with disjunctions are much less common than those solely composed of conjunctions. Hence we speculate even more complex interaction forms are uncommon.

We intend to explore each of these issues in future studies.

5. RELATED WORK

There are several threads of related work.

Interaction Discovery. As discussed earlier, Reisner et al. [22] use the symbolic executor Otter to fully explore the configuration space of a software system and extract interactions from the resulting information. In that work, interactions were limited to conjunctions, while iGen supports much richer interaction templates. Moreover, while symbolic

execution is powerful, it suffers major limitations. First, experience shows it has limited scalability. For example, the Otter experiments required several days on a large cluster and analyzed only a few programs. In contrast, Table 2 shows that iGen is much more efficient. Second, symbolic executors are language-specific, e.g., Otter could not be applied to C++ or Haskell. In contrast, the only language-specific tools iGen relies on are code coverage tools, which are easy to use and widely available for many languages. Finally, symbolic execution is very hard to apply to programs that use frameworks, libraries, and/or native code. Typically symbolic execution users must replace these parts of a system with painstakingly developed "stub" code that implements the functionality in a more symbolic executor-friendly way. Needless to say this process is time-consuming, error-prone, and hard to maintain as systems evolve.

As also discussed earlier, to address some limitations of Otter we developed iTree [27], which uses dynamic analysis and machine learning techniques to generate a set of configurations that achieve high coverage. iTree works by constructing an "interaction tree," where each node of the tree is a formula, and conjoining the formulae on a path from the root to a node yields a potential interaction. However, while iTree does infer some interactions, its main goal is to achieve high coverage. As a result, as we saw in Section 4.2, iTree does not actually infer very many, or very useful, interactions. Moreover, as with Otter, iTree's interactions are limited to conjunctions, although iTree has some support for membership-like constraints to improve efficiency.

Feature Interactions and Presence Conditions. The concepts of *feature interactions* and *presence conditions* are similar to our use of *interactions*. Thüm et al [29] classify problems in software product line research and surveys existing static analysis to solve them. Our use of interactions belongs to the feature-based classification, and we propose a new dynamic analysis to generate them. Apel et al [1] study the number of feature interactions in a system and their effects, including bug triggering, power consumption, etc. Our work complements these results by studying interactions that affect line or expression coverage. Lillack et al [15] use (language-specific) taint analysis to find interactions in Android applications. In contrast, iGen uses a dynamic analysis that is language agnostic (but potentially unsound). Nadi et al [17] and von Rhein et al [30] propose tools that work with presence conditions that are already provided. In contrast, iGen discovers interactions. Czarnecki and Pietroszek [8] check for well-formness errors in UML featured-based model templates using an SAT solver. We intend to explore SMT-based techniques to verify correctness of iGen's inferred interactions.

Combinatorial Interaction Testing. Many researchers have explored combinatorial interaction testing (CIT) [5, 19, 28], a family of techniques for testing a program under a systematically generated set of configurations. One particularly popular approach is called t-way covering arrays, which, given an interaction *strength* t, generate a set of configurations containing all t-way combinations of option settings at least once. Over last 30 years, many studies have focused on improving the speed, quality and flexibility of covering arrays [4, 6, 10, 31, 32]. However, as pointed out in Fouche et al. [13], because developers must choose t a priori and because generating covering arrays quite expensive, developers will often set t to be small, causing higher strength interactions to be ignored.

Invariant Generation. Interactions can also be considered *invariants* that hold at particular locations, i.e., specific option must have specific settings whenever execution reaches that location. Thus, iGen can be seen as a likely invariant generator that works for one particular class of invariants, those restricted to configuration options and with specific forms (quantifier-free expressions involving only equalities and certain conjunctions and disjunctions).

Other researchers have proposed general-purpose invariant generators. Daikon [12] is a well-known dynamic invariant generation system that works by hypothesizing many potential invariants and then using run-time monitoring to eliminate those that do not hold. Daikon includes a large list of invariant templates. DIG is a more recent invariant generator that supports more expressive invariants including nonlinear arithmetic, disjunctive polynomials, and relations among arrays [18]. DySy is another invariant generator that uses symbolic execution for invariant inference [7].

iGen differs from Daikon, DIG, and DySy in three main ways. First, iGen can compute potentially long disjunctive invariants very efficiently (via pointwise union and negation). In contrast, Daikon requires users to provide "splitting" conditions [11] to find disjunctive invariants such as "if c then a else b", and DIG uses complex and expensive-to-compute convex hulls to represent disjunctions. DySy is bounded by the limitations of symbolic execution —the language-specificity and the difficult of analyzing frameworks, libraries, and native code. Second, Daikon, DIG, and DySy do not attempt to search for more executions to refine invariants. Finally, iGen tries to find interactions for every location, while previous work only considers specific locations (e.g., loops and function entrance and exit) to reduce run-time overhead.

6. CONCLUSION

We presented iGen, a new, lightweight approach to infer interactions, which are formulae that describe the configurations covering a location. iGen discovers interactions by running the subject program under a set of configurations to determine coverage information; computing the pointwise union of various sets of covering and non-covering configurations; and then combining the resulting formulae to produce an interaction. iGen repeats this process, iteratively refining the set of configurations, until it no new coverage is achieved and no new interactions are produced. We applied iGen to 29 programs written in five different languages and demonstrated that iGen infers precise interactions; it does so using a small fraction of the number of possible configurations; iGen confirms several observations made by prior work; and the disjunctive and mixed interactions inferred by iGen occur with nontrivial frequency. We believe iGen takes an important step forward in the practical understanding of configurable systems.

Acknowledgments

We thank the anonymous reviewers for their many helpful comments. This research was supported in part by NSF CCF-1116740, CCF-1139021, and CCF-1319666.

7. REFERENCES

[1] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. Exploring feature interactions in the wild: the new feature-interaction challenge. In *International Workshop on Feature-Oriented Software Development*, pages 1–8. ACM, 2013.

[2] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In *International Conference on Automated Software Engineering*, pages 73–82. ACM, 2010.

[3] Bisect; coverage tool for OCaml. http://bisect.x9c.fr, accessed on 2016-03-07.

[4] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960–970, 2006.

[5] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, 1996.

[6] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *International Conference on Software Engineering*, pages 38–48. IEEE, 2003.

[7] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *International Conference on Software Engineering*, pages 281–290. ACM, 2008.

[8] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *International Conference on Generative Programming and Component Engineering*, pages 211–220. ACM, 2006.

[9] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Internal Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[10] G. Demiroz and C. Yilmaz. Cost-aware combinatorial interaction testing. In *International Conference on Advances in System Testing and Validation Lifecycle*, Nov. 2012.

[11] M. D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin. Dynamically discovering program invariants involving collections. Technical report, University of Washington, 2000.

[12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, pages 35–45, 2007.

[13] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *International Symposium on Software Testing and Analysis*, pages 177–188, 2009.

[14] Hpc; coverate tool for Haskell. https://wiki.haskell.org/Haskell_program_coverage, accessed on 2016-03-07.

[15] M. Lillack, C. Kästner, and E. Bodden. Tracking load-time configuration options. In *International Conference on Automated Software Engineering*, pages 445–456. ACM, 2014.

[16] MDevel:Cover; coverage tool for Perl. http://search.cpan.org/ pjcj/Devel-Cover-1.20/lib/Devel/Cover.pm, accessed on 2016-03-07.

[17] S. Nadi, T. Berger, C. Kastner, and K. Czarnecki. Where do configuration constraints stem from? an extraction approach and an empirical study. *Transactions on Software Engineering*, 41(8):820–841, 2015.

[18] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. DIG: a dynamic invariant generator for polynomial and array invariants. *Transactions on Software Engineering and Methodology*, 23(4):30, 2014.

[19] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11, 2011.

[20] Perl Power Tools. http://perlpowertools.com, accessed on 2016-03-07.

[21] Coverage tool for Python. https://wiki.python.org/moin/CodeCoverage, accessed on 2016-03-07.

[22] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *International Conference on Software Engineering*, pages 445–454. ACM, 2010.

[23] V. Rijsbergen. Cj information retrieval. 1979.

[24] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *International Conference on Software Engineering*, pages 461–470. ACM, 2011.

[25] SLOCCount. http://www.dwheeler.com/sloccount, accessed on 2015-03-07.

[26] C. Song, A. Porter, and J. S. Foster. iTree: Efficiently Discovering High-Coverage Configurations Using Interaction Trees. In *International Conference on Software Engineering*, pages 903–913, Zurich, Switzerland, June 2012.

[27] C. Song, A. Porter, and J. S. Foster. iTree: efficiently discovering high-coverage configurations using interaction trees. *Transactions on Software Engineering*, 40(3):251–265, 2014.

[28] K.-C. Tai and Y. Lie. A test generation strategy for pairwise testing. *Transactions on Software Engineering*, 28(1):109–111, 2002.

[29] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis strategies for software product lines. *School of Computer Science, University of Magdeburg, Tech. Rep. FIN-004-2012*, 2012.

[30] A. Von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger. Presence-condition simplification in highly configurable systems. In *International Conference on Software Engineering*, pages 178–188. IEEE Press, 2015.

[31] C. Yilmaz, M. B. Cohen, A. Porter, et al. Covering arrays for efficient fault characterization in complex configuration spaces. *Transactions on Software Engineering*, 32(1):20–34, 2006.

[32] X. Yuan, M. B. Cohen, and A. M. Memon. GUI interaction testing: Incorporating event context. *Transactions on Software Engineering*, 37(4):559–574, 2011.