iGen: Dynamic Interaction Inference for Configurable Software

ThanhVu (Vu) Nguyen*,
Ugur Koc†, Javran Cheng†, Jeffrey S. Foster†, Adam A. Porter†

*University of Nebraska, Lincoln, †University of Maryland, College Park

FSE 2016

Interactions in Configurable Systems

Modern software systems are highly-configurable

- Increases flexibility and add features
- But too many configs complicate many analysis tasks
  - Understanding, testing, debugging, etc
  - How configs affect *line coverage* (the focus of this work)

Interactions in Configurable Systems

Modern software systems are highly-configurable

- Increases flexibility and add features
- But too many configs complicate many analysis tasks
  - Understanding, testing, debugging, etc
  - How configs affect *line coverage* (the focus of this work)

A precise and compact description of configurations is valuable

- Help developers analyze useful info about configs
  - Find important options affecting program coverage
  - Compute minimal set of configs to achieve high coverage
- Discover such a description is possible in practice (not every config leads to different coverage behaviors)

Example

Program with 7 config options

- 6 bools and $z \in \{0, 1, 2, 3, 4\}$
- Config space: 320 configs

s	t	u	v	x	y	z	cov
0	0	0	0	0	0	0	$L2, L3$
			⋮				
1	1	1	1	1	1	3	$L0, L1, L3, L4, L5$

```
//opts: s, t, u, v, x, y, z
int maxz = 3;

if(x && y) {
  printf("L0\n");
  if(!(0 < z && z < maxz))
    printf("L1\n");
}else{
  printf("L2\n");
}

printf("L3\n");
if(u && v) {
  printf("L4\n");
  if(s || t){
    printf("L5\n");
  }
}
```

Example

Program with 7 config options

- 6 bools and $z \in \{0, 1, 2, 3, 4\}$
- Config space: 320 configs

s	t	u	v	x	y	z	cov
0	0	0	0	0	0	0	$L2, L3$
			$\vdots$				
1	1	1	1	1	1	3	$L0, L1, L3, L4, L5$

Use interactions to describe config space

- $x \wedge y$: L0
- $x \wedge y \wedge z \in \{0, 3, 4\}$: L1
- $\overline{x} \vee \overline{y}$: L2
- $u \wedge v$: L4
- $(u \wedge v) \wedge (s \vee t)$: L5

```
//opts: s, t, u, v, x, y, z
int maxz = 3;

if(x && y) {
  printf("L0\n");
  if(!(0 < z && z < maxz))
    printf("L1\n");
}else{
  printf("L2\n");
}

printf("L3\n");
if(u && v) {
  printf("L4\n");
  if(s || t){
    printf("L5\n");
  }
}
```

Example

Program with 7 config options

- 6 bools and $z \in \{0, 1, 2, 3, 4\}$
- Config space: 320 configs

s	t	u	v	x	y	z	cov
0	0	0	0	0	0	0	$L2, L3$
			$\vdots$				
1	1	1	1	1	1	3	$L0, L1, L3, L4, L5$

Use interactions to describe config space

- $x \wedge y$: L0
- $x \wedge y \wedge z \in \{0, 3, 4\}$: L1
- $\overline{x} \vee \overline{y}$: L2
- $u \wedge v$: L4
- $(u \wedge v) \wedge (s \vee t)$: L5

Interaction templates: *conj* $(x \wedge y)$, *disj* $(\overline{x} \vee \overline{y})$, *mixed* $(u \wedge v) \wedge (s \vee t)$

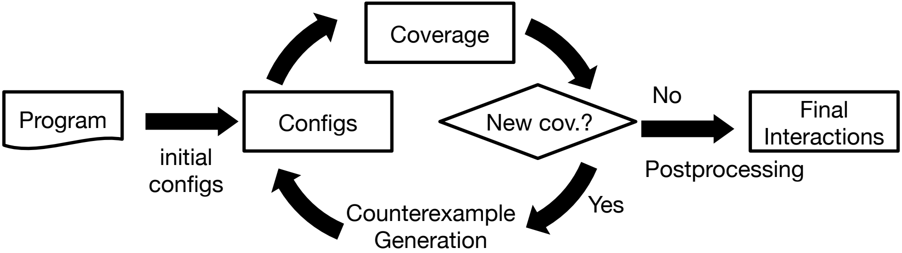
```
//opts: s, t, u, v, x, y, z
int maxz = 3;

if(x && y) {
  printf("L0\n");
  if(!(0 < z && z < maxz))
    printf("L1\n");
}else{
  printf("L2\n");
}

printf("L3\n");
if(u && v) {
  printf("L4\n");
  if(s || t){
    printf("L5\n");
  }
}
```

Contributions

iGen: a dynamic approach to finding interactions wrt line coverage

- Focus on options having finite domains, e.g., boolean, $\{0, 64, 128\}$
- Scale to large, highly-configurable systems, e.g., `httpd`: $\geq 2^{50}$ configs
- Language independent, e.g., tested on programs written in C, Perl, Python, OCaml, Haskell
- Work in presence of framework, libraries, and native code

iGen: Overview



- Run program on a set of initial configs, obtain cov info
- For each covered location, infer interactions
- Inferred results may be imprecise (insufficient data), thus create new (counterexamples) configs to refine interactions
- Repeat until can no longer find new interactions or refine existing ones

Demonstration

Interactions

- $x \wedge y$: L0
- $x \wedge y \wedge z \in \{0, 3, 4\}$: L1
- $\overline{x} \vee \overline{y}$: L2
- $u \wedge v$: L4
- $(u \wedge v) \wedge (s \vee t)$: L5

```
//opts: s, t, u, v, x, y, z
int maxz = 3;

if(x && y) {
  printf("L0\n");
  if(!(0 < z && z < maxz))
    printf("L1\n");
}else{
  printf("L2\n");
}

printf("L3\n");
if(u && v) {
  printf("L4\n");
  if(s || t){
    printf("L5\n");
  }
}
```

Initial Configurations

Create initial configs having each option value used at least once and obtain cov info

- Contains all possible settings of each individual option
- E.g., all 5 values $\{0, 1, 2, 3, 4\}$ of z are used

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	$L2, L3, L4, L5$

For each loc, infer interactions using different *templates*, e.g., *conj* $(x \wedge y)$, *disj* $(\overline{x} \vee \overline{y})$, *mixed* $(u \wedge v) \wedge (s \vee t)$

Conjunctive Interactions

The conj template

- *conjunctions* of membership constraints e.g., $x \in \{1\} \wedge y \in \{1\} \wedge z \in \{0, 3, 4\}$ for $L1$

```
int maxz = 3;
if(x && y) {
  ...
  if(!(0 < z && z < maxz))
    printf("L1\n");
  ...
}
```

Conjunctive Interactions

The conj template

- *conjunctions* of membership constraints e.g., $x \in \{1\} \wedge y \in \{1\} \wedge z \in \{0, 3, 4\}$ for $L1$

```
int maxz = 3;
if(x && y) {
  ...
  if(!(0 < z && z < maxz))
    printf("L1\n");
  ...
}
```

- Use pointwise union $\uplus$ to compute *conj* over configs
  1. Union the values for each option, e.g., $s = \{0, 1\} = 0 \vee 1 = \top$

config	s	t	u	v	x	y	z	coverage
c_2	1	1	0	0	1	1	0	$L0, L1, L3$
c_4	0	0	1	1	1	1	3	$L0, L1, L3, L4$
union	$\top$	$\top$	$\top$	$\top$	1	1	0, 3	

  2. Conjoin the unions to get $x \wedge y \wedge z \in \{0, 3\}$

Conjunctive Interactions

The conj template

- *conjunctions* of membership constraints e.g., $x \in \{1\} \wedge y \in \{1\} \wedge z \in \{0, 3, 4\}$ for $L1$

```
int maxz = 3;
if(x && y) {
  ...
  if(!(0 < z && z < maxz))
    printf("L1\n");
  ...
}
```

- Use pointwise union $\uplus$ to compute *conj* over configs
  1. Union the values for each option, e.g., $s = \{0, 1\} = 0 \vee 1 = \top$

config	s	t	u	v	x	y	z	coverage
c_2	1	1	0	0	1	1	0	$L0, L1, L3$
c_4	0	0	1	1	1	1	3	$L0, L1, L3, L4$
union	$\top$	$\top$	$\top$	$\top$	1	1	0, 3	

  2. Conjoin the unions to get $x \wedge y \wedge z \in \{0, 3\}$

To infer *conj* for a loc: apply $\uplus$ to configs covering that loc

- For $L1$, $c_2 \uplus c_4 = x \wedge y \wedge z \in \{0, 3\}$

Conjunctive Interactions

The conj template

- *conjunctions* of membership constraints e.g., $x \in \{1\} \wedge y \in \{1\} \wedge z \in \{0,3,4\}$ for $L1$

```
int maxz = 3;
if(x && y) {
  ...
  if(!(0 < z && z < maxz))
    printf("L1\n");
  ...
}
```

- Use pointwise union $\Cup$ to compute *conj* over configs
  1. Union the values for each option, e.g., $s = \{0,1\} = 0 \vee 1 = \top$

config	s	t	u	v	x	y	z	coverage
c_2	1	1	0	0	1	1	0	$L0, L1, L3$
c_4	0	0	1	1	1	1	3	$L0, L1, L3, L4$
union	$\top$	$\top$	$\top$	$\top$	1	1	0, 3	

  2. Conjoin the unions to get $x \wedge y \wedge z \in \{0,3\}$

To infer *conj* for a loc: apply $\Cup$ to configs covering that loc

- For $L1$, $c_2 \Cup c_4 = x \wedge y \wedge z \in \{0,3\}$
- Almost, but not quite right $(x \wedge y \wedge z \in \{0,3,4\})$

Interaction Refinement

For $L1$, $conj = x \wedge y \wedge z \in \{0, 3\}$

- Need more configs
- E.g., a config having $z = 4$ covering $L1$

```
int maxz = 3;
if(x && y) {
  ...
  if(!(0 < z && z < maxz))
    printf("L1\n");
}
...
```

Interaction Refinement

For $L1$, $conj = x \wedge y \wedge z \in \{0, 3\}$

- Need more configs
- E.g., a config having $z = 4$ covering $L1$

```
int maxz = 3;
if(x && y) {
  ...
  if(!(0 < z && z < maxz))
    printf("L1\n");
}
...
```

Idea: create new configs to refine existing results

- Select an existing int for some loc
- Systematically change int to create *potentially counterexample* configs (cex's)

Interaction Refinement

For $L1$, $conj = x \wedge y \wedge z \in \{0, 3\}$

- Need more configs
- E.g., a config having $z = 4$ covering $L1$

```
int maxz = 3;
if(x && y) {
  ...
  if(!(0 < z && z < maxz))
    printf("L1\n");
}
...
```

Idea: create new configs to refine existing results

- Select an existing int for some loc
- Systematically change int to create *potentially counterexample* configs (cex's)

Intuition: if cex's, which are different than int, can still cover `loc`, then can use them to refine *int*.

Interaction Refinement: Example

- Pick an existing int to refine, e.g., $conj = x \wedge y \wedge z \in \{0, 3\}$ for $L1$

Interaction Refinement: Example

- Pick an existing int to refine, e.g., $conj = x \wedge y \wedge z \in \{0, 3\}$ for $L1$
- Systematically change $conj$ to create cex's

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	$L0, L1, L3$

  - Each cex disagrees with $conj = x \wedge y \wedge z \in \{0, 3\}$ in exactly one setting, e.g., c_6 has $x = 0$, c_7 has $y = 0$, and c_8 has $z = 1$, ..
  - Create random settings for other options (e.g., s, t, u, v)

Interaction Refinement: Example

- Pick an existing int to refine, e.g., $conj = x \wedge y \wedge z \in \{0, 3\}$ for $L1$
- Systematically change $conj$ to create cex's

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	$L0, L1, L3$

  - Each cex disagrees with $conj = x \wedge y \wedge z \in \{0, 3\}$ in exactly one setting, e.g., c_6 has $x = 0$, c_7 has $y = 0$, and c_8 has $z = 1$, ..
  - Create random settings for other options (e.g., s, t, u, v)

- Next iteration, applying $\uplus$ to c_2, c_4, c_{10} yields $x \wedge y \wedge z \in \{0, 3, 4\}$ (the correct interaction for $L1$)

Disjunctive Interactions

The disj template

- E.g., $\overline{x} \vee \overline{y}$ for $L2$
- Cannot apply $\uplus$ directly (get *conj*, not *disj*)

```
...
if(x && y) {
  printf("L0\n");
  ...
} else{
  printf("L2\n");
}
```

Disjunctive Interactions

The disj template

- E.g., $\overline{x} \vee \overline{y}$ for $L2$
- Cannot apply $\uplus$ directly (get *conj*, not *disj*)

```
...
if(x && y) {
  printf("L0\n");
  ...
} else{
  printf("L2\n");
}
```

Intuition:

- Every `loc` is either covered or *not-covered* by a config
- *Complement* of *non-covering* configs $\equiv$ covering configs

Disjunctive Interactions

The disj template

- E.g., $\overline{x} \vee \overline{y}$ for $L2$
- Cannot apply $\cup$ directly (get *conj*, not *disj*)

```
...
if(x && y) {
  printf("L0\n");
  ...
} else{
  printf("L2\n");
}
```

Intuition:

- Every `loc` is either covered or *not-covered* by a config
- *Complement* of *non-covering* configs $\equiv$ covering configs

Idea: apply $\cup$ on *non-covering* configs and negate

Disjunctive Interactions: Example

To infer *disj* for $L2$

- Obtain configs c_2 and c_4 that do not cover $L2$

config	s	t	u	v	x	y	z	coverage
c_2	1	1	0	0	1	1	0	$L0, L1, L3$
c_4	0	0	1	1	1	1	3	$L0, L1, L3, L4$
union	$\top$	$\top$	$\top$	$\top$	1	1	$0, 3$	

```
...
if(x && y) {
  printf("L0\n");
  ...
} else{
  printf("L2\n");
}
```

- Compute $c_2 \uplus c_4$ to get $conj' = x \wedge y \wedge z \in \{0, 3, 4\}$
- Negate $conj'$ to get $disj = \overline{x} \vee \overline{y} \vee z \in \{1, 2\}$ for $L2$
- (At the end) Check that *disj* actually satisfies all configs covering $L2$

Disjunctive Interactions: Example

To infer *disj* for $L2$

- Obtain configs c_2 and c_4 that do not cover $L2$

config	s	t	u	v	x	y	z	coverage
c_2	1	1	0	0	1	1	0	$L0, L1, L3$
c_4	0	0	1	1	1	1	3	$L0, L1, L3, L4$
union	$\top$	$\top$	$\top$	$\top$	1	1	$0, 3$	

```
...
if(x && y) {
  printf("L0\n");
  ...
} else{
  printf("L2\n");
}
```

- Compute $c_2 \uplus c_4$ to get $conj' = x \wedge y \wedge z \in \{0, 3, 4\}$
- Negate $conj'$ to get $disj = \overline{x} \vee \overline{y} \vee z \in \{1, 2\}$ for $L2$
- (At the end) Check that *disj* actually satisfies all configs covering $L2$

$\uplus$ + negation: straightforward extension of *conj* inference

- Subsequent iterations create cex's to refine *conj'* and thus *disj*
- Also used to compute mixed interactions, e.g., $u \wedge v \wedge (s \vee t)$ for $L5$

Experiments

- iGen is implemented in Python and uses Z3 to simplify formulae
- 29 subject programs:
  - 9 GNU and 8 Powertool `coreutils` (e.g., `cat`, `ln`, `ls`), 10 various progs (e.g., `gzip`, `pandoc`, `httpd`), 2 progs from prev work (`vsftp`, `ngircd`)
  - 5 Languages: C, Perl, Python, OCaml, Haskell
  - Locs: 25 - 250K
  - Options: 2 - 50 binary or finite-domain valued
  - Config spaces: 4 to 1.1×10^{15} possible configs
  - Test suites: default tests (if available) + manually created tests
- Evaluation
  - Medians over 21 runs for each program (randomness due to creating initial and new configs)
  - Tested on 2.4 Ghz Intel Xeon, 16 GB RAM, Linux

Correctness: Does iGen produce correct interactions?

Comparing iGen's iterative algorithm to exhaustive run

- Obtain *"ground truths"*
  - Create *all* possible configs for 14 programs with smallest config space
  - Use existing symbolic execution info for `vsftpd` and `ngircd`
- Results: iGen produces similar coverage (missed 3 lines) and interaction (92% similarity) results comparing ground truths

Correctness: Does iGen produce correct interactions?

Comparing iGen's iterative algorithm to exhaustive run

- Obtain *"ground truths"*
  - Create *all* possible configs for 14 programs with smallest config space
  - Use existing symbolic execution info for `vsftpd` and `ngircd`
- Results: iGen produces similar coverage (missed 3 lines) and interaction (92% similarity) results comparing ground truths

Manual Inspection on iGen's results

- Identify several interactions involving *all* options
- Discover mismatched behaviors, e.g., GNU `uname` and Perl `uname`

Efficiency: How does iGen perform?

Scale well to large programs

- Use a small fraction of total config space, e.g., `httpd`: $838/10^{15}$
- Much faster than prev symbolic exec work, e.g., `vsftp`, `ngircd`: an hr vs 2 weeks

prog	cspace	configs	time (s)
id	1,024	157	34
uname	2,048	95	15
cat	4,096	131	42
mv	5,120	106	38
ln	10,240	213	96
date	17,280	680	350
join	18,432	323	158
sort	6,291,456	1346	3113
ls	3.5e+14	2175	9837
p-id	256	82	283
p-uname	64	28	62
p-cat	128	26	246
p-ln	4	4	42
p-date	3,360	160	2061
p-join	4,608	111	1573
p-sort	2,048	116	3947
p-ls	6.7e7	272	13803
cloc	524,288	210	5017
ack	4.3e+9	1347	23127
grin	2,097,152	242	411
pylint	5.8e+10	1916	27175
hlint	8,192	328	9525
pandoc	4.0e+9	653	23515
unison	393,216	381	4641
bibtex2html	1.2e+9	670	667
gzip	131,072	495	12029
httpd	1.1e+15	838	197390
vsftpd	2.1e+9	620	652
ngircd	29,764	650	1469

Analysis: What can we learn from iGen's results?

- Interactions are rare (far less than number of possible *ints*)
  - E.g., iGen discovers 4 *ints* for `p-cat`, which has 4373 possible *ints*
  - Overall $\leq 0.1\%$ than number of possible interactions

Analysis: What can we learn from iGen's results?

- Interactions are rare (far less than number of possible *ints*)
  - E.g., iGen discovers 4 *ints* for `p-cat`, which has 4373 possible *ints*
  - Overall $\leq 0.1\%$ than number of possible interactions
- Longer `conj` tend to be built on shorter `conj`
  - E.g., if $a \wedge b \wedge c \wedge d$ is an interaction, then $a \wedge b$ is also likely an *int* (potentially due to nested guards)
  - For most programs, conj of length ≥ 3 include a shorter int

Analysis: What can we learn from iGen's results?

- Interactions are rare (far less than number of possible *ints*)
  - E.g., iGen discovers 4 *ints* for `p-cat`, which has 4373 possible *ints*
  - Overall $\leq 0.1\%$ than number of possible interactions
- Longer `conj` tend to be built on shorter `conj`
  - E.g., if $a \wedge b \wedge c \wedge d$ is an interaction, then $a \wedge b$ is also likely an *int* (potentially due to nested guards)
  - For most programs, conj of length ≥ 3 include a shorter int
- Most cov achieved by shorter *ints*, but longer *ints* needed for max cov
  - 87% of coverage is obtained by `ints` of length less than 3
  - 5 programs (`id, uname, cat, p-join, httpd`) have interactions involving all options

Analysis: What can we learn from iGen's results?

- Many *enabling* options (options set in certain ways for high cov), e.g.,
  - `vsftpd`, disabling ssl and local and enabling anon are important to cov
  - `httpd` requires both -enable-http and -enable-so (shared modules)

Analysis: What can we learn from iGen's results?

- Many *enabling* options (options set in certain ways for high cov), e.g.,
  - `vsftpd`, disabling ssl and local and enabling anon are important to cov
  - `httpd` requires both -enable-http and -enable-so (shared modules)
- Disjunctive and mixed interactions are required
  - Appear in in 26/29 benchmark programs
  - Approx 20% of inferred interactions are *disj* and *mixed* interactions

Analysis: Minimal Covering Configs

Minimal covering configs

- Use inferred interactions to compute small sets of configs achieving full cov found by iGen
- E.g., only 2/320 configs needed to cover all lines $L0 - L5$ in example
- Develop a *greedy algorithm* that combines compatible interactions to create high-cov configurations

Result: achieve very small minimal config sets

Analysis: Minimal Covering Configs

Minimal covering configs

- Use inferred interactions to compute small sets of configs achieving full cov found by iGen
- E.g., only 2/320 configs needed to cover all lines $L0 - L5$ in example
- Develop a *greedy algorithm* that combines compatible interactions to create high-cov configurations

Result: achieve very small minimal config sets

prog	cspace	min configs
id	1024	10
uname	2048	4
cat	4096	6
mv	5120	4
ln	10240	7
date	17280	7
join	18432	7
sort	6291456	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	524288	6
ack	4.3×10^{9}	13
grin	2097152	6
pylint	5.8×10^{10}	11
hlint	8192	5
pandoc	4.0×10^{9}	5
unison	393216	7
bibtex2html	1.2×10^{9}	8
gzip	131072	10
httpd	1.1×10^{15}	5
vsftpd	2.1×10^{9}	6
ngircd	29764	6

Summary

iGen: a light weight approach to analyze interactions

- Use dynamic analysis to infer interactions
- Generate new (cex) configurations to refine results
- Efficiently compute different kind of interactions (*conj*, *disj*, *mixed*)

Summary

iGen: a light weight approach to analyze interactions

- Use dynamic analysis to infer interactions
- Generate new (cex) configurations to refine results
- Efficiently compute different kind of interactions (*conj*, *disj*, *mixed*)

Evaluation

- Work on highly-configurable software systems in a variety of languages
- Infer precise interactions using a very small number of configs
- Confirm hypotheses about configurable software
    - Config space can be effectively described a small number of *ints*
    - Longer *conj*'s often built on shorter ones
    - Most cov achieved by shorter *ints*, but longer *ints* needed for max cov
    - Enabling options and expressive interactions are necessary