Exact Gap Computation for Code Coverage Metrics

Dirk Richter, Christian Berg

7th Workshop on Model-Based Testing
March 25, 2012

Martin-Luther-Universität Halle-Wittenberg
Naturwissenschaftliche Fakultät III, Institut für Informatik,
Lehrstuhl Softwaretechnik und Programmiersprachen,
http://swt.informatik.uni-halle.de/

Motivation

- test generation + test data selection difficult
- code coverage metrics: estimate quality of test suites
- coverage 100% $\rightarrow$ quality fine
- practice: 100% impossible (e.g. dead code)
- tester improve test suite: Is adding an extra test wise?
- **Question: What is the maximal possible coverage?**
- presented here: framework to answer this exactly

Motivation

- test generation + test data selection difficult
- code coverage metrics: estimate quality of test suites
- coverage 100% $\rightarrow$ quality fine
- practice: 100% impossible (e.g. dead code)
- tester improve test suite: Is adding an extra test wise?
- **Question: What is the maximal possible coverage?**
- presented here: framework to answer this exactly

Motivation

- test generation + test data selection difficult
- code coverage metrics: estimate quality of test suites
- coverage 100% → quality fine
- practice: 100% impossible (e.g. dead code)
- tester improve test suite: Is adding an extra test wise?
- **Question: What is the maximal possible coverage?**
- presented here: framework to answer this exactly

Motivation

- test generation + test data selection difficult
- code coverage metrics: estimate quality of test suites
- coverage 100% $\rightarrow$ quality fine
- practice: 100% impossible (e.g. dead code)
- tester improve test suite: Is adding an extra test wise?
- **Question: What is the maximal possible coverage?**
- presented here: framework to answer this exactly

Motivation

- test generation + test data selection difficult
- code coverage metrics: estimate quality of test suites
- coverage 100% $\rightarrow$ quality fine
- practice: 100% impossible (e.g. dead code)
- tester improve test suite: Is adding an extra test wise?
- **Question: What is the maximal possible coverage?**
- presented here: framework to answer this exactly
  - based on: model checking techniques

    ISO-C compatible semantic

Motivation

- test generation + test data selection difficult
- code coverage metrics: estimate quality of test suites
- coverage 100% → quality fine
- practice: 100% impossible (e.g. dead code)
- tester improve test suite: Is adding an extra test wise?
- **Question: What is the maximal possible coverage?**
- presented here: framework to answer this exactly
  - based on: model checking techniques
  - ISO C compatible semantic

Motivation

- test generation + test data selection difficult
- code coverage metrics: estimate quality of test suites
- coverage 100% → quality fine
- practice: 100% impossible (e.g. dead code)
- tester improve test suite: Is adding an extra test wise?
- **Question: What is the maximal possible coverage?**
- presented here: framework to answer this exactly
  - based on: model checking techniques
  - ISO-C compatible semantic

Motivation

- test generation + test data selection difficult
- code coverage metrics: estimate quality of test suites
- coverage 100% $\rightarrow$ quality fine
- practice: 100% impossible (e.g. dead code)
- tester improve test suite: Is adding an extra test wise?
- **Question: What is the maximal possible coverage?**
- presented here: framework to answer this exactly
  - based on: model checking techniques
  - ISO-C compatible semantic

Motivation

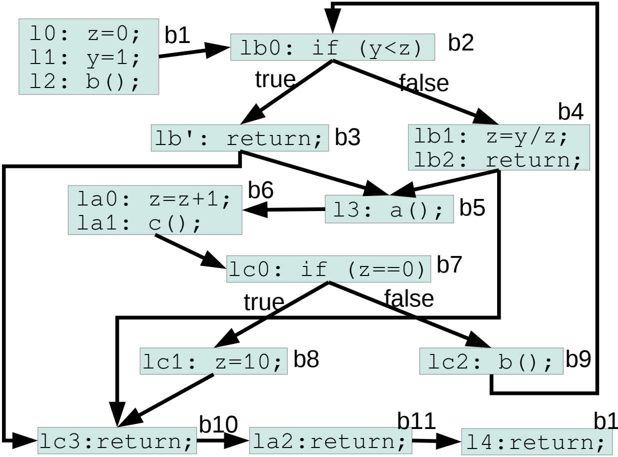
- test generation + test data selection difficult
- code coverage metrics: estimate quality of test suites
- coverage 100% → quality fine
- practice: 100% impossible (e.g. dead code)
- tester improve test suite: Is adding an extra test wise?
- **Question: What is the maximal possible coverage?**
- presented here: framework to answer this exactly
    - based on: model checking techniques
    - ISO-C compatible semantic

```
char y,z;
void b() {
 lb0: if(y<z)return
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: if (z == 0)
        lc1: z = 10;
   else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```



Example P_1 in ISO-C syntax and corresponding BBI-CFG

nodes: *blocks*(P), *edges*(P) : interprocedural control flow

Code Coverage Metrics γ

- test suite $t \in T_P$: set of tests $t = \{\alpha_1, \alpha_2, \ldots\}$ for program P
- code coverage metric $\gamma : T_P \rightarrow [0,1]$ monotonically increasing
- function coverage $\gamma_f^P(t) := |func(t)|/|func(P)|$
- statement coverage $\gamma_s^P(t) := |stats(t)|/|stats(P)|$
- decision coverage $\gamma_d^P(t) := |edges(t)|/|edges(P)|$
- branch coverage $\gamma_b^P(t) := |blocks(t)|/|blocks(P)|$

- $bExpr(l) = \{e_1, e_2, \ldots, e_n\}$ ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \bullet l \in labels(P), e \in bExpr(l)\}$
- condition or predicate coverage metric: $\gamma_c^P(t) := |exval(t, P)|/(2 \cdot |BExpr(P)|)$

Code Coverage Metrics γ

- test suite $t \in T_P$: set of tests $t = \{\alpha_1, \alpha_2, ...\}$ for program P
- code coverage metric $\gamma : T_P \rightarrow [0,1]$ monotonically increasing
- function coverage $\gamma_f^P(t) := |func(t)|/|func(P)|$
- statement coverage $\gamma_s^P(t) := |stats(t)|/|stats(P)|$
- decision coverage $\gamma_d^P(t) := |edges(t)|/|edges(P)|$
- branch coverage $\gamma_b^P(t) := |blocks(t)|/|blocks(P)|$

- $bExpr(l) = \{e_1, e_2, \ldots, e_n\}$ ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \bullet l \in labels(P), e \in bExpr(l)\}$
- condition or predicate coverage metric: $\gamma_c^P(t) := |exval(t, P)|/(2 \cdot |BExpr(P)|)$

Code Coverage Metrics γ

- test suite $t \in T_P$: set of tests $t = \{\alpha_1, \alpha_2, \ldots\}$ for program P
- code coverage metric $\gamma : T_P \rightarrow [0,1]$ monotonically increasing
- function coverage $\gamma_f^P(t) := |func(t)|/|func(P)|$
- statement coverage $\gamma_s^P(t) := |stats(t)|/|stats(P)|$
- decision coverage $\gamma_d^P(t) := |edges(t)|/|edges(P)|$
- branch coverage $\gamma_b^P(t) := |blocks(t)|/|blocks(P)|$

- $bExpr(l) = \{e_1, e_2, \ldots, e_n\}$ ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \bullet l \in labels(P), e \in bExpr(l)\}$
- condition or predicate coverage metric: $\gamma_c^P(t) := |exval(t, P)|/(2 \cdot |BExpr(P)|)$

Code Coverage Metrics γ

- test suite $t \in T_P$: set of tests $t = \{\alpha_1, \alpha_2, ...\}$ for program P
- code coverage metric $\gamma : T_P \rightarrow [0, 1]$ monotonically increasing
- function coverage $\gamma_f^P(t) := |func(t)|/|func(P)|$
- statement coverage $\gamma_s^P(t) := |stats(t)|/|stats(P)|$
- decision coverage $\gamma_d^P(t) := |edges(t)|/|edges(P)|$
- branch coverage $\gamma_b^P(t) := |blocks(t)|/|blocks(P)|$

- $bExpr(l) = \{e_1, e_2, \dots e_n\}$ ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \bullet l \in labels(P), e \in bExpr(l)\}$
- condition or predicate coverage metric:
  $\gamma_c^P(t) := |exval(t, P)|/(2 \cdot |BExpr(P)|)$

Code Coverage Metrics γ

- test suite $t \in T_P$: set of tests $t = \{\alpha_1, \alpha_2, ...\}$ for program P
- code coverage metric $\gamma : T_P \to [0,1]$ monotonically increasing
- function coverage $\gamma_f^P(t) := |func(t)|/|func(P)|$
- statement coverage $\gamma_s^P(t) := |stats(t)|/|stats(P)|$
- decision coverage $\gamma_d^P(t) := |edges(t)|/|edges(P)|$
- branch coverage $\gamma_b^P(t) := |blocks(t)|/|blocks(P)|$

- $bExpr(l) = \{e_1, e_2, \ldots e_n\}$ ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \bullet l \in labels(P), e \in bExpr(l)\}$
- condition or predicate coverage metric: $\gamma_c^P(t) := |exval(t, P)|/(2 \cdot |BExpr(P)|)$

Code Coverage Metrics γ

- test suite $t \in T_P$: set of tests $t = \{\alpha_1, \alpha_2, ...\}$ for program P
- code coverage metric $\gamma : T_P \to [0,1]$ monotonically increasing
- function coverage $\gamma_f^P(t) := |func(t)|/|func(P)|$
- statement coverage $\gamma_s^P(t) := |stats(t)|/|stats(P)|$
- decision coverage $\gamma_d^P(t) := |edges(t)|/|edges(P)|$
- branch coverage $\gamma_b^P(t) := |blocks(t)|/|blocks(P)|$

- $bExpr(l) = \{e_1, e_2, \ldots e_n\}$ ... set of all boolean sub-expressions
- $BExpr(P) := \{(l,e) \bullet l \in labels(P), e \in bExpr(l)\}$
- condition or predicate coverage metric: $\gamma_c^P(t) := |exval(t,P)|/(2 \cdot |BExpr(P)|)$

Code Coverage Metrics γ

- test suite $t \in T_P$: set of tests $t = \{\alpha_1, \alpha_2, ...\}$ for program P
- code coverage metric $\gamma : T_P \to [0,1]$ monotonically increasing
- function coverage $\gamma_f^P(t) := |func(t)|/|func(P)|$
- statement coverage $\gamma_s^P(t) := |stats(t)|/|stats(P)|$
- decision coverage $\gamma_d^P(t) := |edges(t)|/|edges(P)|$
- branch coverage $\gamma_b^P(t) := |blocks(t)|/|blocks(P)|$

- $bExpr(l) = \{e_1, e_2, \ldots e_n\}$ ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \bullet l \in labels(P), e \in bExpr(l)\}$
- condition or predicate coverage metric: $\gamma_c^P(t) := |exval(t, P)|/(2 \cdot |BExpr(P)|)$

Code Coverage Metrics γ

- test suite $t \in T_P$: set of tests $t = \{\alpha_1, \alpha_2, ...\}$ for program P
- code coverage metric $\gamma : T_P \rightarrow [0,1]$ monotonically increasing
- function coverage $\gamma_f^P(t) := |func(t)|/|func(P)|$
- statement coverage $\gamma_s^P(t) := |stats(t)|/|stats(P)|$
- decision coverage $\gamma_d^P(t) := |edges(t)|/|edges(P)|$
- branch coverage $\gamma_b^P(t) := |blocks(t)|/|blocks(P)|$

- $bExpr(l) = \{e_1, e_2, \ldots e_n\}$ ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \bullet l \in labels(P), e \in bExpr(l)\}$
- condition or predicate coverage metric: $\gamma_c^P(t) := |exval(t, P)|/(2 \cdot |BExpr(P)|)$

Code Coverage Metrics γ

- test suite $t \in T_P$: set of tests $t = \{\alpha_1, \alpha_2, ...\}$ for program P
- code coverage metric $\gamma : T_P \rightarrow [0,1]$ monotonically increasing
- function coverage $\gamma_f^P(t) := |func(t)|/|func(P)|$
- statement coverage $\gamma_s^P(t) := |stats(t)|/|stats(P)|$
- decision coverage $\gamma_d^P(t) := |edges(t)|/|edges(P)|$
- branch coverage $\gamma_b^P(t) := |blocks(t)|/|blocks(P)|$

- $bExpr(l) = \{e_1, e_2, \ldots e_n\}$ ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \bullet l \in labels(P), e \in bExpr(l)\}$
- condition or predicate coverage metric: $\gamma_c^P(t) := |exval(t, P)|/(2 \cdot |BExpr(P)|)$

Code Coverage Metric Gap δ

- let $\gamma : T_P \rightarrow [0, 1]$ be a code coverage metric
- **code coverage metric gap** $\delta_\gamma(P) := \inf_{t \in T_P}(1 - \gamma(t))$
- smallest diff.: practical coverage ratio vs. theoretical maximum
- not computable in Turing powerful languages
- more expressive model $\rightarrow$ more accurate computat. of gap δ_γ

$\rightarrow$ adequate modeling

Code Coverage Metric Gap δ

- let $\gamma : T_P \to [0, 1]$ be a code coverage metric
- **code coverage metric gap** $\delta_\gamma(P) := \inf_{t \in T_P}(1 - \gamma(t))$
- smallest diff.: practical coverage ratio vs. theoretical maximum
- not computable in Turing powerful languages
- more expressive model $\to$ more accurate computat. of gap δ_γ

$\to$ adequate modeling

Code Coverage Metric Gap δ

- let $\gamma : T_P \to [0,1]$ be a code coverage metric
- **code coverage metric gap** $\delta_\gamma(P) := \inf_{t \in T_P}(1 - \gamma(t))$
- smallest diff.: practical coverage ratio vs. theoretical maximum
- not computable in Turing powerful languages
- more expressive model $\to$ more accurate computat. of gap δ_γ

$\to$ adequate modeling

Code Coverage Metric Gap δ

- let $\gamma : T_P \rightarrow [0, 1]$ be a code coverage metric
- **code coverage metric gap** $\delta_\gamma(P) := \inf_{t \in T_P}(1 - \gamma(t))$
- smallest diff.: practical coverage ratio vs. theoretical maximum
- not computable in Turing powerful languages
- more expressive model $\rightarrow$ more accurate computat. of gap δ_γ

$\rightarrow$ adequate modeling

Code Coverage Metric Gap δ

- let $\gamma : T_P \to [0,1]$ be a code coverage metric
- **code coverage metric gap** $\delta_\gamma(P) := \inf_{t \in T_P}(1 - \gamma(t))$
- smallest diff.: practical coverage ratio vs. theoretical maximum
- not computable in Turing powerful languages
- more expressive model $\to$ more accurate computat. of gap δ_γ

$\to$ adequate modeling

Code Coverage Metric Gap δ

- let $\gamma : T_P \to [0,1]$ be a code coverage metric
- **code coverage metric gap** $\delta_\gamma(P) := \inf_{t \in T_P}(1 - \gamma(t))$
- smallest diff.: practical coverage ratio vs. theoretical maximum
- not computable in Turing powerful languages
- more expressive model $\to$ more accurate computat. of gap δ_γ

$\to$ adequate modeling

Suitable Models (basic statements)

- symbolic pushdown system(SPDS): ISO-C compatible semantic
- describe SPDS by ISO-C syntax (platform-specific semantic) $e \in Expr$, "$l : s$" $\in stats$, s has the following forms:
  - $x[e_1] = e_2$; writing $[\![e_2]\!]$ into variable x at index $[\![e_1]\!]$
  - $f(x_1, \ldots, x_n)$; function call (call by value)
  - *return*; function return
  - *if* (e) *goto* l'; conditional jump to $l' \in labels$
- not Turing powerful (infinite Kripke structure)

Suitable Models (basic statements)

- symbolic pushdown system(SPDS): ISO-C compatible semantic
- describe SPDS by ISO-C syntax (platform-specific semantic)
  $e \in Expr$, "$l : s$" $\in stats$, s has the following forms:
  - $x[e_1] = e_2$; writing $[\![e_2]\!]$ into variable x at index $[\![e_1]\!]$
  - $f(x_1, \ldots, x_n)$; function call (call by value)
  - *return*; function return
  - *if* (e) *goto* l'; conditional jump to $l' \in labels$
- not Turing powerful (infinite Kripke structure)

Suitable Models (basic statements)

- symbolic pushdown system(SPDS): ISO-C compatible semantic
- describe SPDS by ISO-C syntax (platform-specific semantic) $e \in Expr$, "$l : s$" $\in stats$, s has the following forms:
  - $x[e_1] = e_2;$ writing $[\![e_2]\!]$ into variable x at index $[\![e_1]\!]$
  - $f(x_1, \ldots, x_n);$ function call (call by value)
  - $return;$ function return
  - $if\ (e)\ goto\ l';$ conditional jump to $l' \in labels$
- not Turing powerful (infinite Kripke structure)

Suitable Models (basic statements)

- symbolic pushdown system(SPDS): ISO-C compatible semantic
- describe SPDS by ISO-C syntax (platform-specific semantic) $e \in Expr$, "$l : s$" $\in stats$, s has the following forms:
  - $x[e_1] = e_2$; writing $[\![e_2]\!]$ into variable x at index $[\![e_1]\!]$
  - $f(x_1, \ldots, x_n)$; function call (call by value)
  - *return*; function return
  - *if* (e) *goto* l'; conditional jump to $l' \in labels$
- not Turing powerful (infinite Kripke structure)

Suitable Models (basic statements)

- symbolic pushdown system(SPDS): ISO-C compatible semantic
- describe SPDS by ISO-C syntax (platform-specific semantic) $e \in Expr$, "$l : s$" $\in stats$, s has the following forms:
  - $x[e_1] = e_2$; writing $[\![e_2]\!]$ into variable x at index $[\![e_1]\!]$
  - $f(x_1, \ldots, x_n)$; function call (call by value)
  - *return*; function return
  - *if* (e) *goto* l'; conditional jump to $l' \in labels$
- not Turing powerful (infinite Kripke structure)

Suitable Models (basic statements)

- symbolic pushdown system(SPDS): ISO-C compatible semantic
- describe SPDS by ISO-C syntax (platform-specific semantic) $e \in Expr$, "$l : s$" $\in stats$, s has the following forms:
  - $x[e_1] = e_2;$ writing $[\![e_2]\!]$ into variable x at index $[\![e_1]\!]$
  - $f(x_1, \ldots, x_n);$ function call (call by value)
  - *return*; function return
  - *if* (e) *goto* l'; conditional jump to $l' \in labels$
- not Turing powerful (infinite Kripke structure)

Suitable Models (basic statements)

- symbolic pushdown system(SPDS): ISO-C compatible semantic
- describe SPDS by ISO-C syntax (platform-specific semantic)
  $e \in Expr$, "$l : s$" $\in stats$, s has the following forms:
  - $x[e_1] = e_2;$ writing $[\![e_2]\!]$ into variable x at index $[\![e_1]\!]$
  - $f(x_1, \ldots, x_n);$ function call (call by value)
  - *return*; function return
  - *if* (e) *goto* l'; conditional jump to $l' \in labels$
- not Turing powerful (infinite Kripke structure)

Symbolic Pushdown System (SPDS)

- SPDS S: global variables $vgbl$, functions $func$, $main \in func$
- variable x (array) has integer type $bits(x)$ and length $len(x)$
- each statement has unique label $l \in labels(S)$
- $fst(f)$ is first label of function f
- state of S: configuration $(g, [(l, c)\ldots])$ current execute label l
- $g : vgbl \times \mathbb{Z} \to \mathbb{Z}$, $c : vlcl(l) \times \mathbb{Z} \to \mathbb{Z}$ ...variable settings
- $g(x, i), c(x, i)$ current value of variable x on index i

Symbolic Pushdown System (SPDS)

- SPDS S: global variables $vgbl$, functions $func$, $main \in func$
- variable x (array) has integer type $bits(x)$ and length $len(x)$
- each statement has unique label $l \in labels(S)$
- $fst(f)$ is first label of function f
- state of S: configuration $(g, [(l, c)...])$ current execute label l
- $g : vgbl \times \mathbb{Z} \to \mathbb{Z}, \quad c : vlcl(l) \times \mathbb{Z} \to \mathbb{Z}$ ...variable settings
- $g(x, i), c(x, i)$ current value of variable x on index i

Symbolic Pushdown System (SPDS)

- SPDS S: global variables $vgbl$, functions $func$, $main \in func$
- variable x (array) has integer type $bits(x)$ and length $len(x)$
- each statement has unique label $l \in labels(S)$
- $fst(f)$ is first label of function f
- state of S: configuration $(g, [(l, c)...])$ current execute label l
- $g : vgbl \times \mathbb{Z} \to \mathbb{Z}, \quad c : vlcl(l) \times \mathbb{Z} \to \mathbb{Z}$ ...variable settings
- $g(x, i), c(x, i)$ current value of variable x on index i

Symbolic Pushdown System (SPDS)

- SPDS S: global variables $vgbl$, functions $func$, $main \in func$
- variable x (array) has integer type $bits(x)$ and length $len(x)$
- each statement has unique label $l \in labels(S)$
- $fst(f)$ is first label of function f
- state of S: configuration $(g, [(l, c)...])$ current execute label l
- $g : vgbl \times \mathbb{Z} \rightarrow \mathbb{Z}$, $c : vlcl(l) \times \mathbb{Z} \rightarrow \mathbb{Z}$ ...variable settings
- $g(x, i), c(x, i)$ current value of variable x on index i

Symbolic Pushdown System (SPDS)

- SPDS S: global variables $vgbl$, functions $func$, $main \in func$
- variable x (array) has integer type $bits(x)$ and length $len(x)$
- each statement has unique label $l \in labels(S)$
- $fst(f)$ is first label of function f
- state of S: configuration $(g, [(l, c)...])$ current execute label l
- $g : vgbl \times \mathbb{Z} \to \mathbb{Z}$, $c : vlcl(l) \times \mathbb{Z} \to \mathbb{Z}$ ...variable settings
- $g(x, i), c(x, i)$ current value of variable x on index i

Symbolic Pushdown System (SPDS)

- SPDS S: global variables $vgbl$, functions $func$, $main \in func$
- variable x (array) has integer type $bits(x)$ and length $len(x)$
- each statement has unique label $l \in labels(S)$
- $fst(f)$ is first label of function f
- state of S: configuration $(g, [(l, c)...])$ current execute label l
- $g : vgbl \times \mathbb{Z} \to \mathbb{Z}$, $c : vlcl(l) \times \mathbb{Z} \to \mathbb{Z}$ ...variable settings
- $g(x, i), c(x, i)$ current value of variable x on index i

Symbolic Pushdown System (SPDS)

- SPDS S: global variables $vgbl$, functions $func$, $main \in func$
- variable x (array) has integer type $bits(x)$ and length $len(x)$
- each statement has unique label $l \in labels(S)$
- $fst(f)$ is first label of function f
- state of S: configuration $(g, [(l, c)...])$ current execute label l
- $g : vgbl \times \mathbb{Z} \to \mathbb{Z}$, $c : vlcl(l) \times \mathbb{Z} \to \mathbb{Z}$ ...variable settings
- $g(x, i), c(x, i)$ current value of variable x on index i

Exact Gap Computation Framework

Given: program P + code coverage metric $\gamma : T_P \to [0,1]$

1. Create SPDS S with ISO-C compatible semantic for P
2. Modify S to S' to enable gap analysis
3. Compute exact variable ranges for the new variables in S'
4. Conclude exact size of the gap $\delta_\gamma(\mathbf{S})$
5. Conclude size of the gap $\delta_\gamma(\mathbf{P})$

Exact Gap Computation Framework

Given: program P + code coverage metric $\gamma : T_P \to [0, 1]$

1. Create SPDS S with ISO-C compatible semantic for P
2. Modify S to S' to enable gap analysis
3. Compute exact variable ranges for the new variables in S'
4. Conclude exact size of the gap $\delta_\gamma(\mathbf{S})$
5. Conclude size of the gap $\delta_\gamma(\mathbf{P})$

Exact Gap Computation Framework

Given: program P + code coverage metric $\gamma : T_P \to [0,1]$

1. Create SPDS S with ISO-C compatible semantic for P
2. Modify S to S' to enable gap analysis
3. Compute exact variable ranges for the new variables in S'
4. Conclude exact size of the gap $\delta_\gamma(\mathbf{S})$
5. Conclude size of the gap $\delta_\gamma(\mathbf{P})$

Exact Gap Computation Framework

Given: program P + code coverage metric $\gamma : T_P \rightarrow [0,1]$

1. Create SPDS S with ISO-C compatible semantic for P
2. Modify S to S' to enable gap analysis
3. Compute exact variable ranges for the new variables in S'
4. Conclude exact size of the gap $\delta_\gamma(\mathbf{S})$
5. Conclude size of the gap $\delta_\gamma(\mathbf{P})$

Exact Gap Computation Framework

Given: program P + code coverage metric $\gamma : T_P \rightarrow [0, 1]$

1. Create SPDS S with ISO-C compatible semantic for P
2. Modify S to S' to enable gap analysis
3. Compute exact variable ranges for the new variables in S'
4. Conclude exact size of the gap $\delta_\gamma(\mathbf{S})$
5. Conclude size of the gap $\delta_\gamma(\mathbf{P})$

Step 1 - SPDS Modeling

- map ISO-C statements to basic statements (abbreviations)
- other languages similar (e.g. Java using JMoped)
- higher level concepts can be simulated... (see paper)
  - omit returns and labels: introduced during interpretation expressions as parameters: evaluate to temporary variables return expressions: new global variable ret_f for function f function call in expression intermediate compiler representation further concepts: unconditional jump, skip statement, random numbers, conditional statements local variable definitions, loops (do, while, for), modular arithmetic/integer overflow dynamic memory, pointers, call by reference, dynamic arrays
- recursion bounded: call by reference + local variables in heap

Step 1 - SPDS Modeling

- map ISO-C statements to basic statements (abbreviations)
- other languages similar (e.g. Java using JMoped)
- higher level concepts can be simulated... (see paper)
  - omit returns and labels: introduced during interpretation
  - expressions as parameters: evaluate to temporary variables
  - return expressions: new global variable ret_f for function f
  - function call in expression intermediate compiler representation
  - further concepts: unconditional jump, skip statement, random numbers, conditional statements local variable definitions, loops (do, while, for), modular arithmetic/integer overflow, dynamic memory, pointers, call by reference, dynamic arrays
- recursion bounded: call by reference + local variables in heap

Step 1 - SPDS Modeling

- map ISO-C statements to basic statements (abbreviations)
- other languages similar (e.g. Java using JMoped)
- higher level concepts can be simulated... (see paper)
  - omit returns and labels: introduced during interpretation
  - expressions as parameters: evaluate to temporary variables
  - return expressions: new global variable $iret_f$ for function f
  - function call in expression:intermediate compiler representation
  - further concepts: unconditional jump, skip statement, random numbers, conditional statements local variable definitions, loops (do, while, for), modular arithmetic/integer overflow, dynamic memory, pointers, call by reference, dynamic arrays
- recursion bounded: call by reference + local variables in heap

Step 1 - SPDS Modeling

- map ISO-C statements to basic statements (abbreviations)
- other languages similar (e.g. Java using JMoped)
- higher level concepts can be simulated... (see paper)
  - omit returns and labels: introduced during interpretation
  - expressions as parameters: evaluate to temporary variables
  - return expressions: new global variable $iret_f$ for function f
  - function call in expression:intermediate compiler representation
  - further concepts: unconditional jump, skip statement, random numbers, conditional statements local variable definitions, loops (do, while, for), modular arithmetic/integer overflow, dynamic memory, pointers, call by reference, dynamic arrays
- recursion bounded: call by reference + local variables in heap

Step 1 - SPDS Modeling

- map ISO-C statements to basic statements (abbreviations)
- other languages similar (e.g. Java using JMoped)
- higher level concepts can be simulated... (see paper)
  - omit returns and labels: introduced during interpretation
  - expressions as parameters: evaluate to temporary variables
  - return expressions: new global variable $iret_f$ for function f
  - function call in expression:intermediate compiler representation
  - further concepts: unconditional jump, skip statement, random numbers, conditional statements local variable definitions, loops (do, while, for), modular arithmetic/integer overflow, dynamic memory, pointers, call by reference, dynamic arrays
- recursion bounded: call by reference + local variables in heap

Step 1 - SPDS Modeling

- map ISO-C statements to basic statements (abbreviations)
- other languages similar (e.g. Java using JMoped)
- higher level concepts can be simulated... (see paper)
  - omit returns and labels: introduced during interpretation
  - expressions as parameters: evaluate to temporary variables
  - return expressions: new global variable $iret_f$ for function f
  - function call in expression:intermediate compiler representation
  - further concepts: unconditional jump, skip statement, random numbers, conditional statements local variable definitions, loops (do, while, for), modular arithmetic/integer overflow, dynamic memory, pointers, call by reference, dynamic arrays
- recursion bounded: call by reference + local variables in heap

Step 1 - SPDS Modeling

- map ISO-C statements to basic statements (abbreviations)
- other languages similar (e.g. Java using JMoped)
- higher level concepts can be simulated... (see paper)
  - omit returns and labels: introduced during interpretation
  - expressions as parameters: evaluate to temporary variables
  - return expressions: new global variable $iret_f$ for function f
  - function call in expression:intermediate compiler representation
  - further concepts: unconditional jump, skip statement, random numbers, conditional statements local variable definitions, loops (do, while, for), modular arithmetic/integer overflow, dynamic memory, pointers, call by reference, dynamic arrays
- recursion bounded: call by reference + local variables in heap

Step 1 - SPDS Modeling

- map ISO-C statements to basic statements (abbreviations)
- other languages similar (e.g. Java using JMoped)
- higher level concepts can be simulated... (see paper)
  - omit returns and labels: introduced during interpretation
  - expressions as parameters: evaluate to temporary variables
  - return expressions: new global variable $iret_f$ for function f
  - function call in expression:intermediate compiler representation
  - further concepts: unconditional jump, skip statement, random numbers, conditional statements local variable definitions, loops (do, while, for), modular arithmetic/integer overflow, dynamic memory, pointers, call by reference, dynamic arrays
- recursion bounded: call by reference + local variables in heap

Step 1 - SPDS Modeling

- map ISO-C statements to basic statements (abbreviations)
- other languages similar (e.g. Java using JMoped)
- higher level concepts can be simulated... (see paper)
  - omit returns and labels: introduced during interpretation
  - expressions as parameters: evaluate to temporary variables
  - return expressions: new global variable $iret_f$ for function f
  - function call in expression:intermediate compiler representation
  - further concepts: unconditional jump, skip statement, random numbers, conditional statements local variable definitions, loops (do, while, for), modular arithmetic/integer overflow, dynamic memory, pointers, call by reference, dynamic arrays
- recursion bounded: call by reference + local variables in heap

Step 3 - Extraction of Exact Variable Ranges

- model checker Moped: algorithm to create $Post^*$ automaton
- $Post^*$ accepts reachable configurations (infinite set)
- map SPDS S to Remopla (input language of Moped)

→ exact variable $range_l(x) = \{[\![x]\!]^c_g \bullet (g, [(l, c)...]) \in Post^*(S)\}$

- problem: $Post^*(S)$ infinite

Step 3 - Extraction of Exact Variable Ranges

- model checker Moped: algorithm to create $Post^*$ automaton
- $Post^*$ accepts reachable configurations (infinite set)
- map SPDS S to Remopla (input language of Moped)

→ exact variable $range_l(x) = \{[\![x]\!]_g^c \bullet (g, [(l, c)...]) \in Post^*(S)\}$

- problem: $Post^*(S)$ infinite

Step 3 - Extraction of Exact Variable Ranges

- model checker Moped: algorithm to create $Post^*$ automaton
- $Post^*$ accepts reachable configurations (infinite set)
- map SPDS S to Remopla (input language of Moped)

→ exact variable $range_l(x) = \{ [\![x]\!]_g^c \bullet (g, [(l, c)...]) \in Post^*(S) \}$

- problem: $Post^*(S)$ infinite

Step 3 - Extraction of Exact Variable Ranges

- model checker Moped: algorithm to create $Post^*$ automaton
- $Post^*$ accepts reachable configurations (infinite set)
- map SPDS S to Remopla (input language of Moped)

→ exact variable $range_l(x) = \{[\![x]\!]_g^c \bullet (g, [(l, c)...]) \in Post^*(S)\}$

- problem: $Post^*(S)$ infinite

Step 3 - Extraction of Exact Variable Ranges

- model checker Moped: algorithm to create $Post^*$ automaton
- $Post^*$ accepts reachable configurations (infinite set)
- map SPDS S to Remopla (input language of Moped)

→ exact variable $range_l(x) = \{[\![x]\!]_g^c \bullet (g, [(l, c)...]) \in Post^*(S)\}$

- problem: $Post^*(S)$ infinite

Step 3 - problem: $Post^*(S)$ infinite

- solution: restrict on reachable heads $h(S) := \{(g, (l, c)) \ldots\}$

$\rightarrow$ finite, because of finite variable types

- implementation: symbolical computation of...
  1. characteristic function $q : \{0,1\}^n \rightarrow \{0,1\}$ for $h(S)$ using OBDD restrict operation out of $Post^*$
  2. characteristic function $r : \{0,1\}^m \rightarrow \{0,1\}$ for $range(x)$ using OBDD co-factorization out of $h(S)$

Step 3 - problem: $Post^*(S)$ infinite

- solution: restrict on reachable heads $h(S) := \{(g, (l, c)) \dots\}$

$\rightarrow$ finite, because of finite variable types

- implementation: symbolical computation of...
  1. characteristic function $q : \{0,1\}^n \rightarrow \{0,1\}$ for $h(S)$ using OBDD restrict operation out of $Post^*$
  2. characteristic function $r : \{0,1\}^m \rightarrow \{0,1\}$ for $range_l(x)$ using OBDD co-factorization out of $h(S)$

Step 3 - problem: $Post^*(S)$ infinite

- solution: restrict on reachable heads $h(S) := \{(g, (l, c)) \ldots\}$

$\rightarrow$ finite, because of finite variable types

- implementation: symbolical computation of...
  1. characteristic function $q : \{0,1\}^n \rightarrow \{0,1\}$ for $h(S)$ using OBDD restrict operation out of $Post^*$
  2. characteristic function $r : \{0,1\}^m \rightarrow \{0,1\}$ for $range_l(x)$ using OBDD co-factorization out of $h(S)$

Step 3 - problem: $Post^*(S)$ infinite

- solution: restrict on reachable heads $h(S) := \{(g, (l, c)) \ldots\}$

→ finite, because of finite variable types

- implementation: symbolical computation of...
  1. characteristic function $q : \{0, 1\}^n \rightarrow \{0, 1\}$ for $h(S)$ using OBDD restrict operation out of $Post^*$
  2. characteristic function $r : \{0, 1\}^m \rightarrow \{0, 1\}$ for $range_l(x)$ using OBDD co-factorization out of $h(S)$

Step 3 - problem: $Post^*(S)$ infinite

- solution: restrict on reachable heads $h(S) := \{(g, (l, c)) \ldots\}$

$\rightarrow$ finite, because of finite variable types

- implementation: symbolical computation of...
  1. characteristic function $q : \{0,1\}^n \rightarrow \{0,1\}$ for $h(S)$ using OBDD restrict operation out of $Post^*$
  2. characteristic function $r : \{0,1\}^m \rightarrow \{0,1\}$ for $range_l(x)$ using OBDD co-factorization out of $h(S)$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Function, statement and branch coverage gap $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$

- introduce new global variable $x \notin vgbl(S) \rightsquigarrow$ *SPDS* S'
- $bits(x) = 1, len(x) = 1$
- observation: x has random value $[\![x]\!] \in \{0, 1\}$ on each **reachable** label (function entry/block entry)
- compute exact $range_l(x) \quad \forall l \in labels(S')$
- $range_l(x) \neq \emptyset \iff$ label l reachable in S

$$\Rightarrow \delta_s(S) = 1 - \frac{|\{l{:}s \in stats(S) \bullet range_l^{S'}(x) \neq \emptyset\}|}{|stats(S)|}$$

$$\Rightarrow \delta_f(S) = 1 - \frac{|\{f \in func(S) \bullet range_{fst(f)}^{S'}(x) \neq \emptyset\}|}{|func(S)|}$$

$$\Rightarrow \delta_b(S) = 1 - \frac{|\{b \in blocks(S) \bullet range_{fst(b)}^{S'}(x) \neq \emptyset\}|}{|blocks(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Function, statement and branch coverage gap $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$

- introduce new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $bits(x) = 1, len(x) = 1$
- observation: x has random value $[\![x]\!] \in \{0,1\}$ on each **reachable** label (function entry/block entry)
- compute exact $range_l(x) \quad \forall l \in labels(S')$
- $range_l(x) \neq \emptyset \;\Leftrightarrow\;$ label l reachable in S

$$\Rightarrow \delta_s(S) = 1 - \frac{|\{l{:}s \in stats(S) \bullet range_l^{S'}(x) \neq \emptyset\}|}{|stats(S)|}$$

$$\Rightarrow \delta_f(S) = 1 - \frac{|\{f \in func(S) \bullet range_{fst(f)}^{S'}(x) \neq \emptyset\}|}{|func(S)|}$$

$$\Rightarrow \delta_b(S) = 1 - \frac{|\{b \in blocks(S) \bullet range_{fst(b)}^{S'}(x) \neq \emptyset\}|}{|blocks(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Function, statement and branch coverage gap $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$

- introduce new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $bits(x) = 1, len(x) = 1$
- observation: x has random value $[\![x]\!] \in \{0,1\}$ on each **reachable** label (function entry/block entry)
- compute exact $range_l(x) \quad \forall l \in labels(S')$
- $range_l(x) \neq \emptyset \;\Leftrightarrow\;$ label l reachable in S

$$\Rightarrow \delta_s(S) = 1 - \frac{|\{l{:}s{\in}stats(S) \bullet range_l^{S'}(x){\neq}\emptyset\}|}{|stats(S)|}$$

$$\Rightarrow \delta_f(S) = 1 - \frac{|\{f{\in}func(S) \bullet range_{fst(f)}^{S'}(x){\neq}\emptyset\}|}{|func(S)|}$$

$$\Rightarrow \delta_b(S) = 1 - \frac{|\{b{\in}blocks(S) \bullet range_{fst(b)}^{S'}(x){\neq}\emptyset\}|}{|blocks(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Function, statement and branch coverage gap $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$

- introduce new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $bits(x) = 1, len(x) = 1$
- observation: x has random value $[\![x]\!] \in \{0,1\}$ on each **reachable** label (function entry/block entry)
- compute exact $range_l(x) \quad \forall l \in labels(S')$
- $range_l(x) \neq \emptyset \quad \Leftrightarrow \quad$ label l reachable in S

$$\Rightarrow \quad \delta_s(S) = 1 - \frac{|\{l{:}s{\in}stats(S) \bullet range_l^{S'}(x){\neq}\emptyset\}|}{|stats(S)|}$$

$$\Rightarrow \quad \delta_f(S) = 1 - \frac{|\{f{\in}func(S) \bullet range_{fst(f)}^{S'}(x){\neq}\emptyset\}|}{|func(S)|}$$

$$\Rightarrow \quad \delta_b(S) = 1 - \frac{|\{b{\in}blocks(S) \bullet range_{fst(b)}^{S'}(x){\neq}\emptyset\}|}{|blocks(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Function, statement and branch coverage gap $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$

- introduce new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $bits(x) = 1, len(x) = 1$
- observation: x has random value $[\![x]\!] \in \{0,1\}$ on each **reachable** label (function entry/block entry)
- compute exact $range_l(x) \quad \forall l \in labels(S')$
- $range_l(x) \neq \emptyset \;\Leftrightarrow\;$ label l reachable in S

$$\Rightarrow \delta_s(S) = 1 - \frac{|\{l{:}s{\in}stats(S) \bullet range_l^{S'}(x){\neq}\emptyset\}|}{|stats(S)|}$$

$$\Rightarrow \delta_f(S) = 1 - \frac{|\{f{\in}func(S) \bullet range_{fst(f)}^{S'}(x){\neq}\emptyset\}|}{|func(S)|}$$

$$\Rightarrow \delta_b(S) = 1 - \frac{|\{b{\in}blocks(S) \bullet range_{fst(b)}^{S'}(x){\neq}\emptyset\}|}{|blocks(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Function, statement and branch coverage gap $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$

- introduce new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $bits(x) = 1, len(x) = 1$
- observation: x has random value $[\![x]\!] \in \{0, 1\}$ on each **reachable** label (function entry/block entry)
- compute exact $range_l(x) \quad \forall l \in labels(S')$
- $range_l(x) \neq \emptyset \;\Leftrightarrow\;$ label l reachable in S

$$\Rightarrow \delta_s(S) = 1 - \frac{|\{l{:}s{\in}stats(S) \bullet range_l^{S'}(x){\neq}\emptyset\}|}{|stats(S)|}$$

$$\Rightarrow \delta_f(S) = 1 - \frac{|\{f{\in}func(S) \bullet range_{fst(f)}^{S'}(x){\neq}\emptyset\}|}{|func(S)|}$$

$$\Rightarrow \delta_b(S) = 1 - \frac{|\{b{\in}blocks(S) \bullet range_{fst(b)}^{S'}(x){\neq}\emptyset\}|}{|blocks(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Function, statement and branch coverage gap $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$

- introduce new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $bits(x) = 1, len(x) = 1$
- observation: x has random value $[\![x]\!] \in \{0,1\}$ on each **reachable** label (function entry/block entry)
- compute exact $range_l(x) \quad \forall l \in labels(S')$
- $range_l(x) \neq \emptyset \ \Leftrightarrow$ label l reachable in S

$$\Rightarrow \delta_s(S) = 1 - \frac{|\{l{:}s{\in}stats(S) \bullet range_l^{S'}(x){\neq}\emptyset\}|}{|stats(S)|}$$

$$\Rightarrow \delta_f(S) = 1 - \frac{|\{f{\in}func(S) \bullet range_{fst(f)}^{S'}(x){\neq}\emptyset\}|}{|func(S)|}$$

$$\Rightarrow \delta_b(S) = 1 - \frac{|\{b{\in}blocks(S) \bullet range_{fst(b)}^{S'}(x){\neq}\emptyset\}|}{|blocks(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Function, statement and branch coverage gap $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$

- introduce new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $bits(x) = 1, len(x) = 1$
- observation: x has random value $[\![x]\!] \in \{0,1\}$ on each **reachable** label (function entry/block entry)
- compute exact $range_l(x) \quad \forall l \in labels(S')$
- $range_l(x) \neq \emptyset \ \Leftrightarrow$ label l reachable in S

$$\Rightarrow \delta_s(S) = 1 - \frac{|\{l{:}s{\in}stats(S)\ \bullet\ range_l^{S'}(x){\neq}\emptyset\}|}{|stats(S)|}$$

$$\Rightarrow \delta_f(S) = 1 - \frac{|\{f{\in}func(S)\ \bullet\ range_{fst(f)}^{S'}(x){\neq}\emptyset\}|}{|func(S)|}$$

$$\Rightarrow \delta_b(S) = 1 - \frac{|\{b{\in}blocks(S)\ \bullet\ range_{fst(b)}^{S'}(x){\neq}\emptyset\}|}{|blocks(S)|}$$

Step 2+4 - $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: if(y<z)return
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: if (z == 0)
          lc1: z = 10;
   else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- new variable x with $bits(x) = len(x) = 1$
- $l2$: call function b
- $lb0$: $(y < z) \rightsquigarrow false$
- $lb1$: division-by-zero

$\rightarrow$ $range_l^{S'}(x) = \emptyset \iff l \notin \{l0, l1, l2, lb0, lb1\}$

$\Rightarrow$ $\delta_s(S) = 1 - \frac{|\{l:s \in stats(S) \bullet range_l^{S'}(x) \neq \emptyset\}|}{|stats(S)|} = 67\%$

$\Rightarrow$ $\delta_f(S) = 1 - \frac{|\{f \in func(S) \bullet range_{fst(f)}^{S'}(x) \neq \emptyset\}|}{|func(S)|} = 50\%$

$\Rightarrow$ $\delta_b(S) = 1 - \frac{|\{b \in blocks(S) \bullet range_{fst(b)}^{S'}(x) \neq \emptyset\}|}{|blocks(S)|} = 75\%$

- no test suite t with branch coverage $\gamma_b^S(t) > 25\%$

Step 2+4 - $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: if(y<z)return
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: if (z == 0)
         lc1: z = 10;
   else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- new variable x with $bits(x) = len(x) = 1$
- $l2$: call function b
- $lb0$: $(y < z) \rightsquigarrow false$
- $lb1$: division-by-zero

$\rightarrow$ $range_l^{S'}(x) = \emptyset \Leftrightarrow l \notin \{l0, l1, l2, lb0, lb1\}$

$$\Rightarrow \delta_s(S) = 1 - \frac{|\{l{:}s \in \boldsymbol{stats}(S) \bullet range_l^{S'}(x) \neq \emptyset\}|}{|\boldsymbol{stats}(S)|} = 67\%$$

$$\Rightarrow \delta_f(S) = 1 - \frac{|\{f \in \boldsymbol{func}(S) \bullet range_{\boldsymbol{fst}(f)}^{S'}(x) \neq \emptyset\}|}{|\boldsymbol{func}(S)|} = 50\%$$

$$\Rightarrow \delta_b(S) = 1 - \frac{|\{b \in \boldsymbol{blocks}(S) \bullet range_{\boldsymbol{fst}(b)}^{S'}(x) \neq \emptyset\}|}{|\boldsymbol{blocks}(S)|} = 75\%$$

- no test suite t with branch coverage $\gamma_b^S(t) > 25\%$

Step 2+4 - $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: if(y<z)return
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: if (z == 0)
         lc1: z = 10;
   else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- new variable x with $bits(x) = len(x) = 1$
- $l2$: call function b
- $lb0$: $(y < z) \rightsquigarrow false$
- $lb1$: division-by-zero

$\rightarrow$ $range_l^{S'}(x) = \emptyset \Leftrightarrow l \notin \{l0, l1, l2, lb0, lb1\}$

$\Rightarrow$ $\delta_s(S) = 1 - \frac{|\{l:s \in stats(S) \bullet range_l^{S'}(x) \neq \emptyset\}|}{|stats(S)|} = 67\%$

$\Rightarrow$ $\delta_f(S) = 1 - \frac{|\{f \in func(S) \bullet range_{fst(f)}^{S'}(x) \neq \emptyset\}|}{|func(S)|} = 50\%$

$\Rightarrow$ $\delta_b(S) = 1 - \frac{|\{b \in blocks(S) \bullet range_{fst(b)}^{S'}(x) \neq \emptyset\}|}{|blocks(S)|} = 75\%$

- no test suite t with branch coverage $\gamma_b^S(t) > 25\%$

Step 2+4 - $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: if(y<z)return
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: if (z == 0)
          lc1: z = 10;
   else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- new variable x with $bits(x) = len(x) = 1$
- $l2$: call function b
- $lb0$: $(y < z) \rightsquigarrow false$
- $lb1$: division-by-zero

$\rightarrow$ $range_l^{S'}(x) = \emptyset \Leftrightarrow l \notin \{l0, l1, l2, lb0, lb1\}$

$$\Rightarrow \delta_s(S) = 1 - \frac{|\{l{:}s \in \boldsymbol{stats}(S) \bullet \boldsymbol{range}_l^{S'}(x) \neq \emptyset\}|}{|\boldsymbol{stats}(S)|} = 67\%$$

$$\Rightarrow \delta_f(S) = 1 - \frac{|\{f \in \boldsymbol{func}(S) \bullet \boldsymbol{range}_{\boldsymbol{fst}(f)}^{S'}(x) \neq \emptyset\}|}{|\boldsymbol{func}(S)|} = 50\%$$

$$\Rightarrow \delta_b(S) = 1 - \frac{|\{b \in \boldsymbol{blocks}(S) \bullet \boldsymbol{range}_{\boldsymbol{fst}(b)}^{S'}(x) \neq \emptyset\}|}{|\boldsymbol{blocks}(S)|} = 75\%$$

- no test suite t with branch coverage $\gamma_b^S(t) > 25\%$

Step 2+4 - $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: if(y<z)return
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: if (z == 0)
         lc1: z = 10;
  else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- new variable x with $bits(x) = len(x) = 1$
- $l2$: call function b
- $lb0$: $(y < z) \rightsquigarrow false$
- $lb1$: division-by-zero

$\rightarrow$ $range_l^{S'}(x) = \emptyset \;\Leftrightarrow\; l \notin \{l0, l1, l2, lb0, lb1\}$

$\Rightarrow$ $\delta_s(S) = 1 - \frac{|\{l : s \in stats(S) \,\bullet\, range_l^{S'}(x) \neq \emptyset\}|}{|stats(S)|} = 67\%$

$\Rightarrow$ $\delta_f(S) = 1 - \frac{|\{f \in func(S) \,\bullet\, range_{fst(f)}^{S'}(x) \neq \emptyset\}|}{|func(S)|} = 50\%$

$\Rightarrow$ $\delta_b(S) = 1 - \frac{|\{b \in blocks(S) \,\bullet\, range_{fst(b)}^{S'}(x) \neq \emptyset\}|}{|blocks(S)|} = 75\%$

- no test suite t with branch coverage $\gamma_b^S(t) > 25\%$

Step 2+4 - $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: if(y<z)return
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: if (z == 0)
          lc1: z = 10;
   else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- new variable x with $bits(x) = len(x) = 1$
- $l2$: call function b
- $lb0$: $(y < z) \rightsquigarrow false$
- $lb1$: division-by-zero

$\rightarrow$ $range_l^{S'}(x) = \emptyset \Leftrightarrow l \notin \{l0, l1, l2, lb0, lb1\}$

$\Rightarrow$ $\delta_s(S) = 1 - \frac{|\{l:s \in stats(S) \bullet range_l^{S'}(x) \neq \emptyset\}|}{|stats(S)|} = 67\%$

$\Rightarrow$ $\delta_f(S) = 1 - \frac{|\{f \in func(S) \bullet range_{fst(f)}^{S'}(x) \neq \emptyset\}|}{|func(S)|} = 50\%$

$\Rightarrow$ $\delta_b(S) = 1 - \frac{|\{b \in blocks(S) \bullet range_{fst(b)}^{S'}(x) \neq \emptyset\}|}{|blocks(S)|} = 75\%$

- no test suite t with branch coverage $\gamma_b^S(t) > 25\%$

Step 2+4 - $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: if(y<z)return
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: if (z == 0)
          lc1: z = 10;
  else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- new variable x with $bits(x) = len(x) = 1$
- $l2$: call function b
- $lb0$: $(y < z) \rightsquigarrow false$
- $lb1$: division-by-zero

$\rightarrow$ $range_l^{S'}(x) = \emptyset \;\Leftrightarrow\; l \notin \{l0, l1, l2, lb0, lb1\}$

$\Rightarrow$ $\delta_s(S) = 1 - \frac{|\{l{:}s \in stats(S) \,\bullet\, range_l^{S'}(x) \neq \emptyset\}|}{|stats(S)|} = 67\%$

$\Rightarrow$ $\delta_f(S) = 1 - \frac{|\{f \in func(S) \,\bullet\, range_{fst(f)}^{S'}(x) \neq \emptyset\}|}{|func(S)|} = 50\%$

$\Rightarrow$ $\delta_b(S) = 1 - \frac{|\{b \in blocks(S) \,\bullet\, range_{fst(b)}^{S'}(x) \neq \emptyset\}|}{|blocks(S)|} = 75\%$

- no test suite t with branch coverage $\gamma_b^S(t) > 25\%$

Step 2+4 - $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: if(y<z)return
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: if (z == 0)
          lc1: z = 10;
   else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- new variable x with $bits(x) = len(x) = 1$
- $l2$: call function b
- $lb0$: $(y < z) \rightsquigarrow false$
- $lb1$: division-by-zero

$\rightarrow$ $range_l^{S'}(x) = \emptyset \;\Leftrightarrow\; l \notin \{l0, l1, l2, lb0, lb1\}$

$\Rightarrow$ $\delta_s(S) = 1 - \frac{|\{l:s\in \mathbf{stats}(S) \,\bullet\, \mathbf{range}_l^{S'}(x)\neq\emptyset\}|}{|\mathbf{stats}(S)|} = 67\%$

$\Rightarrow$ $\delta_f(S) = 1 - \frac{|\{f\in \mathbf{func}(S) \,\bullet\, \mathbf{range}_{\mathbf{fst}(f)}^{S'}(x)\neq\emptyset\}|}{|\mathbf{func}(S)|} = 50\%$

$\Rightarrow$ $\delta_b(S) = 1 - \frac{|\{b\in \mathbf{blocks}(S) \,\bullet\, \mathbf{range}_{\mathbf{fst}(b)}^{S'}(x)\neq\emptyset\}|}{|\mathbf{blocks}(S)|} = 75\%$

- no test suite t with branch coverage $\gamma_b^S(t) > 25\%$

Step 2+4 - $\delta_f(S), \delta_s(S)$ and $\delta_b(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: if(y<z)return
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: if (z == 0)
        lc1: z = 10;
  else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- new variable x with $bits(x) = len(x) = 1$
- $l2$: call function b
- $lb0$: $(y < z) \rightsquigarrow false$
- $lb1$: division-by-zero

$\rightarrow$ $range_l^{S'}(x) = \emptyset \;\Leftrightarrow\; l \notin \{l0, l1, l2, lb0, lb1\}$

$\Rightarrow$ $\delta_s(S) = 1 - \frac{|\{l:s\in stats(S)\ \bullet\ range_l^{S'}(x)\neq\emptyset\}|}{|stats(S)|} = 67\%$

$\Rightarrow$ $\delta_f(S) = 1 - \frac{|\{f\in func(S)\ \bullet\ range_{fst(f)}^{S'}(x)\neq\emptyset\}|}{|func(S)|} = 50\%$

$\Rightarrow$ $\delta_b(S) = 1 - \frac{|\{b\in blocks(S)\ \bullet\ range_{fst(b)}^{S'}(x)\neq\emptyset\}|}{|blocks(S)|} = 75\%$

- no test suite t with branch coverage $\gamma_b^S(t) > 25\%$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Decision coverage gap $\delta_d(S)$

- decision coverage: fraction of edges $(a, b) \in edges(S)$
- remember last executed basic block using ...
- ...new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $len(x) = 1, bits(x) = 1 + \lceil log_2(|blocks(S)|) \rceil$
- each block b has unique number n_b
- $"l : s" \in stats(S) \Rrightarrow "l : x = n_{block(l)}; l' : s;"$

$$\Rightarrow \delta_d(S) = 1 - \frac{|\{(a,b) \in edges(S) \bullet n_a \in range^{S'}_{fst(b)}(x)\}|}{|edges(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Decision coverage gap $\delta_d(S)$

- decision coverage: fraction of edges $(a, b) \in edges(S)$
- remember last executed basic block using ...
- ...new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $len(x) = 1, bits(x) = 1 + \lceil log_2(|blocks(S)|) \rceil$
- each block b has unique number n_b
- $"l : s" \in stats(S) \Rightarrow "l : x = n_{block(l)}; l' : s;"$

$$\Rightarrow \delta_d(S) = 1 - \frac{|\{(a,b) \in edges(S) \bullet n_a \in range^{S'}_{fst(b)}(x)\}|}{|edges(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Decision coverage gap $\delta_d(S)$

- decision coverage: fraction of edges $(a, b) \in edges(S)$
- remember last executed basic block using ...
- ...new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $len(x) = 1, bits(x) = 1 + \lceil log_2(|blocks(S)|) \rceil$
- each block b has unique number n_b
- $"l : s" \in stats(S) \Rightarrow "l : x = n_{block(l)}; l' : s;"$

$$\Rightarrow \delta_d(S) = 1 - \frac{|\{(a,b) \in edges(S) \bullet n_a \in range^{S'}_{fst(b)}(x)\}|}{|edges(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Decision coverage gap $\delta_d(S)$

- decision coverage: fraction of edges $(a, b) \in edges(S)$
- remember last executed basic block using ...
- ...new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $len(x) = 1, bits(x) = 1 + \lceil log_2(|blocks(S)|) \rceil$
- each block b has unique number n_b
- $"l : s" \in stats(S) \Rrightarrow "l : x = n_{block(l)}; l' : s;"$

$$\Rightarrow \delta_d(S) = 1 - \frac{|\{(a,b) \in edges(S) \bullet n_a \in range^{S'}_{fst(b)}(x)\}|}{|edges(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Decision coverage gap $\delta_d(S)$

- decision coverage: fraction of edges $(a,b) \in edges(S)$
- remember last executed basic block using ...
- ...new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $len(x) = 1, bits(x) = 1 + \lceil log_2(|blocks(S)|) \rceil$
- each block b has unique number n_b
- $"l : s" \in stats(S) \Rrightarrow "l : x = n_{block(l)}; l' : s;"$

$$\Rightarrow \delta_d(S) = 1 - \frac{|\{(a,b) \in edges(S) \bullet n_a \in range^{S'}_{fst(b)}(x)\}|}{|edges(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Decision coverage gap $\delta_d(S)$

- decision coverage: fraction of edges $(a, b) \in edges(S)$
- remember last executed basic block using ...
- ...new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $len(x) = 1, bits(x) = 1 + \lceil log_2(|blocks(S)|) \rceil$
- each block b has unique number n_b
- $"l : s" \in stats(S) \Rrightarrow "l : x = n_{block(l)}; l' : s;"$

$$\Rightarrow \delta_d(S) = 1 - \frac{|\{(a,b) \in edges(S) \bullet n_a \in range^{S'}_{fst(b)}(x)\}|}{|edges(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Decision coverage gap $\delta_d(S)$

- decision coverage: fraction of edges $(a, b) \in edges(S)$
- remember last executed basic block using ...
- ...new global variable $x \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $len(x) = 1, bits(x) = 1 + \lceil log_2(|blocks(S)|) \rceil$
- each block b has unique number n_b
- $"l : s" \in stats(S) \Rrightarrow "l : x = n_{block(l)}; l' : s;"$

$$\Rightarrow \delta_d(S) = 1 - \frac{|\{(a,b) \in edges(S) \bullet n_a \in range^{S'}_{fst(b)}(x)\}|}{|edges(S)|}$$

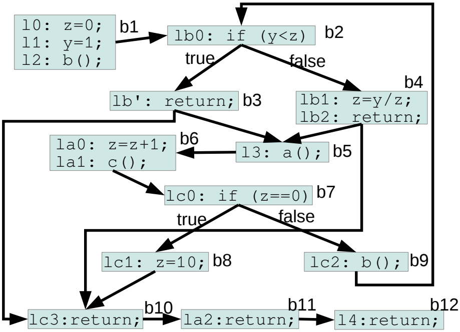
Step 2+4 - Decision coverage gap $\delta_d(S)$ illustration

```
char y,z;
void b() {
 lb0: if(y<z)return
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: if (z == 0)
          lc1: z = 10;
   else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```



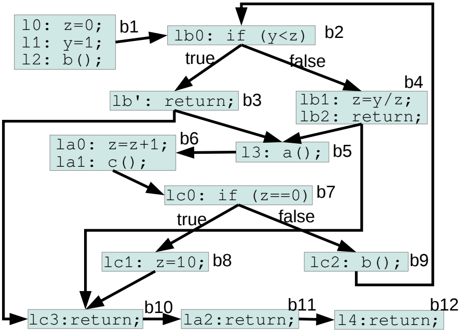
Step 2+4 - Decision coverage gap $\delta_d(S)$ illustration

```
char y,z; int x(4);
void b() {
 lb0: x=2;if(y<z){x=3;return;}
 lb1: x=4; z=y/z;
 lb2: return; }

void c() {
 lc0: x=7; if (z == 0)
         lc1: x=8; z = 10;
   else lc2: x=9; b();
 lc3: x=10; return; }

void a() {
 la0: x=6; z=z+1;
 la1: c();
 la2: x=11; return; }

void main() {
 l0: x=1; z=0;
 l1: y=1;
 l2: b();
 l3: x=5; a();
 l4: x=12; return; }
```



- value of x represents incoming block nr
- exact $rangle_l^{S'}(x)$ indicates incoming blocks
- decision coverage gap $\delta_d(S) = 87\%$

→ only 13% of the CFG edges are coverable

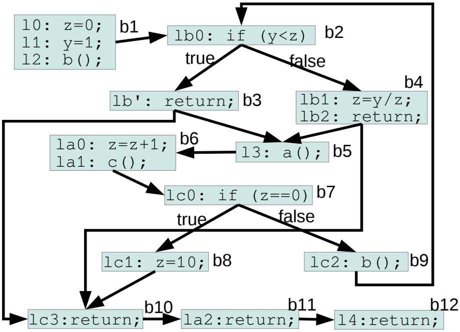
Step 2+4 - Decision coverage gap $\delta_d(S)$ illustration

```
char y,z; int x(4);
void b() {
 lb0: x=2;if(y<z){x=3;return;}
 lb1: x=4; z=y/z;
 lb2: return; }

void c() {
 lc0: x=7; if (z == 0)
         lc1: x=8; z = 10;
   else lc2: x=9; b();
 lc3: x=10; return; }

void a() {
 la0: x=6; z=z+1;
 la1: c();
 la2: x=11; return; }

void main() {
 l0: x=1; z=0;
 l1: y=1;
 l2: b();
 l3: x=5; a();
 l4: x=12; return; }
```



- value of x represents incoming block nr
- exact $rangle_l^{S'}(x)$ indicates incoming blocks
- decision coverage gap $\delta_d(S) = 87\%$

→ only 13% of the CFG edges are coverable

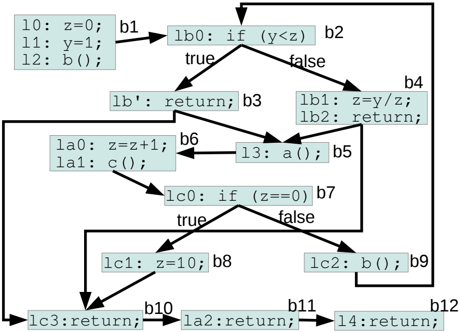
Step 2+4 - Decision coverage gap $\delta_d(S)$ illustration

```
char y,z; int x(4);
void b() {
 lb0: x=2;if(y<z){x=3;return;}
 lb1: x=4; z=y/z;
 lb2: return; }

void c() {
 lc0: x=7; if (z == 0)
          lc1: x=8; z = 10;
  else lc2: x=9; b();
 lc3: x=10; return; }

void a() {
 la0: x=6; z=z+1;
 la1: c();
 la2: x=11; return; }

void main() {
 l0: x=1; z=0;
 l1: y=1;
 l2: b();
 l3: x=5; a();
 l4: x=12; return; }
```



- value of x represents incoming block nr
- exact $rangle_l^{S'}(x)$ indicates incoming blocks
- decision coverage gap $\delta_d(S) = 87\%$

→ only 13% of the CFG edges are coverable

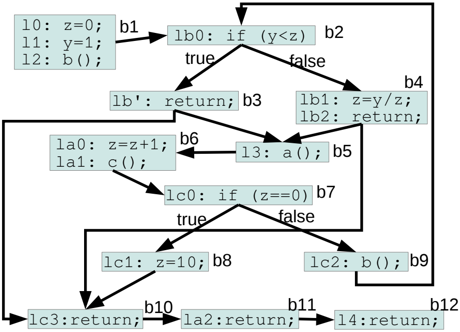
Step 2+4 - Decision coverage gap $\delta_d(S)$ illustration

```
char y,z; int x(4);
void b() {
 lb0: x=2;if(y<z){x=3;return;}
 lb1: x=4; z=y/z;
 lb2: return; }

void c() {
 lc0: x=7; if (z == 0)
          lc1: x=8; z = 10;
   else lc2: x=9; b();
 lc3: x=10; return; }

void a() {
 la0: x=6; z=z+1;
 la1: c();
 la2: x=11; return; }

void main() {
 l0: x=1; z=0;
 l1: y=1;
 l2: b();
 l3: x=5; a();
 l4: x=12; return; }
```



- value of x represents incoming block nr
- exact $rangle_l^{S'}(x)$ indicates incoming blocks
- decision coverage gap $\delta_d(S) = 87\%$

→ only 13% of the CFG edges are coverable

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Condition coverage gap $\delta_c(S)$

- consider every boolean sub-expression e in $SPDS\ S$
- new global variables $x_e \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $len(x_e) = 1, bits(x_e) = 1$
- $"l : s" \quad \Rrightarrow \quad "l : x_{e_1} = e_1; x_{e_2} = e_2; \ldots x_{e_n} = e_n; l' : s;"$
- evaluations of x_{e_i} equivalent to coveraged conditions e_i

$$\Rightarrow \delta_c(S) = 1 - \frac{\sum_{\substack{l \in labels(S) \\ e \in bExpr(l)}} |range_{l'}^{S'}(x_e)|}{2 \cdot |BExpr(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Condition coverage gap $\delta_c(S)$

- consider every boolean sub-expression e in $SPDS\ S$
- new global variables $x_e \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $len(x_e) = 1, bits(x_e) = 1$
- $"l : s" \Rightarrow "l : x_{e_1} = e_1; x_{e_2} = e_2; \ldots x_{e_n} = e_n; l' : s;"$
- evaluations of x_{e_i} equivalent to coveraged conditions e_i

$$\Rightarrow \delta_c(S) = 1 - \frac{\sum_{\substack{l \in labels(S) \\ e \in bExpr(l)}} |range_{l'}^{S'}(x_e)|}{2 \cdot |BExpr(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Condition coverage gap $\delta_c(S)$

- consider every boolean sub-expression e in $SPDS\ S$
- new global variables $x_e \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $len(x_e) = 1, bits(x_e) = 1$
- $"l : s" \Rightarrow "l : x_{e_1} = e_1; x_{e_2} = e_2; \ldots x_{e_n} = e_n; l' : s;"$
- evaluations of x_{e_i} equivalent to coveraged conditions e_i

$$\Rightarrow \delta_c(S) = 1 - \frac{\sum_{\substack{l \in labels(S) \\ e \in bExpr(l)}} |range_{l'}^{S'}(x_e)|}{2 \cdot |BExpr(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Condition coverage gap $\delta_c(S)$

- consider every boolean sub-expression e in $SPDS\ S$
- new global variables $x_e \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $len(x_e) = 1, bits(x_e) = 1$
- $"l : s" \;\Rrightarrow\; "l : x_{e_1} = e_1; x_{e_2} = e_2; \ldots x_{e_n} = e_n; l' : s;"$
- evaluations of x_{e_i} equivalent to coveraged conditions e_i

$$\Rightarrow \delta_c(S) = 1 - \frac{\sum_{\substack{l \in labels(S) \\ e \in bExpr(l)}} |range_{l'}^{S'}(x_e)|}{2 \cdot |BExpr(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Condition coverage gap $\delta_c(S)$

- consider every boolean sub-expression e in *SPDS* S
- new global variables $x_e \notin vgbl(S) \rightsquigarrow$ *SPDS* S'
- $len(x_e) = 1, bits(x_e) = 1$
- $"l : s" \Rrightarrow "l : x_{e_1} = e_1; x_{e_2} = e_2; \ldots x_{e_n} = e_n; l' : s;"$
- evaluations of x_{e_i} equivalent to coveraged conditions e_i

$$\Rightarrow \delta_c(S) = 1 - \frac{\sum_{\substack{l \in labels(S) \\ e \in bExpr(l)}} |range_{l'}^{S'}(x_e)|}{2 \cdot |BExpr(S)|}$$

Step 2+4 - SPDS Supplementation and Exact Gap Inference

Condition coverage gap $\delta_c(S)$

- consider every boolean sub-expression e in $SPDS\ S$
- new global variables $x_e \notin vgbl(S) \rightsquigarrow SPDS\ S'$
- $len(x_e) = 1, bits(x_e) = 1$
- $"l : s" \Rightarrow "l : x_{e_1} = e_1; x_{e_2} = e_2; \ldots x_{e_n} = e_n; l' : s;"$
- evaluations of x_{e_i} equivalent to coveraged conditions e_i

$$\Rightarrow \delta_c(S) = 1 - \frac{\sum_{\substack{l \in labels(S) \\ e \in bExpr(l)}} |range_{l'}^{S'}(x_e)|}{2 \cdot |BExpr(S)|}$$

Step 2+4 - Condition coverage gap $\delta_c(S)$ illustration

```
char y,z;
void b() {
 lb0: if(y<z)return;
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: if (z == 0)
        lc1: z = 10;
   else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

Step 2+4 - Condition coverage gap $\delta_c(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: x=(y<z); p:if(y<z)return;
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: x=(z==0); q:if (z == 0)
           lc1: z = 10;
    else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- exact $rangle_l^{S'}(x) \subseteq \{true, false\}$
- $rangle_p^{S'}(x) = \{false\}$ indicates $y < z$ evals.
- $rangle_q^{S'}(x) = \emptyset$ indicates $z == 0$ evals.

→ condition coverage gap $\delta_c(S) = 75\%$

→ only 25% of the conditions are coverable

Step 2+4 - Condition coverage gap $\delta_c(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: x=(y<z); p:if(y<z)return;
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: x=(z==0); q:if (z == 0)
          lc1: z = 10;
   else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- exact $rangle_l^{S'}(x) \subseteq \{true, false\}$
- $rangle_p^{S'}(x) = \{false\}$ indicates $y < z$ evals.
- $rangle_q^{S'}(x) = \emptyset$ indicates $z == 0$ evals.

→ condition coverage gap $\delta_c(S) = 75\%$

→ only 25% of the conditions are coverable

Step 2+4 - Condition coverage gap $\delta_c(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: x=(y<z); p:if(y<z)return;
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: x=(z==0); q:if (z == 0)
          lc1: z = 10;
   else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- exact $rangle_l^{S'}(x) \subseteq \{true, false\}$
- $rangle_p^{S'}(x) = \{false\}$ indicates $y < z$ evals.
- $rangle_q^{S'}(x) = \emptyset$ indicates $z == 0$ evals.

→ condition coverage gap $\delta_c(S) = 75\%$

→ only 25% of the conditions are coverable

Step 2+4 - Condition coverage gap $\delta_c(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: x=(y<z); p: if (y<z) return;
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: x=(z==0); q: if (z == 0)
          lc1: z = 10;
   else lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- exact $rangle_l^{S'}(x) \subseteq \{true, false\}$
- $rangle_p^{S'}(x) = \{false\}$ indicates $y < z$ evals.
- $rangle_q^{S'}(x) = \emptyset$ indicates $z == 0$ evals.

→ condition coverage gap $\delta_c(S) = 75\%$

→ only 25% of the conditions are coverable

Step 2+4 - Condition coverage gap $\delta_c(S)$ illustration

```
char y,z; bool x;
void b() {
 lb0: x=(y<z); p:if(y<z)return;
 lb1: z=y/z;
 lb2: return; }

void c() {
 lc0: x=(z==0); q:if (z == 0)
         lc1: z = 10;
   else  lc2: b();
 lc3: return; }

void a() {
 la0: z=z+1;
 la1: c();
 la2: return; }

void main() {
 l0: z=0;
 l1: y=1;
 l2: b();
 l3: a();
 l4: return; }
```

- exact $rangle_l^{S'}(x) \subseteq \{true, false\}$
- $rangle_p^{S'}(x) = \{false\}$ indicates $y < z$ evals.
- $rangle_q^{S'}(x) = \emptyset$ indicates $z == 0$ evals.

→ condition coverage gap $\delta_c(S) = 75\%$

→ only 25% of the conditions are coverable

Gap Approximation

- program P equivalent to SPDS S $\rightarrow$ exact gaps
- program P abstracted to SPDS S $\rightarrow$ approximated gaps

- more practical: approximate $range_l(x)$ $\rightarrow$ approximated gaps
- $range_l^+(x)$: over-approximation using data-flow analyzes
- $range_l^-(x)$: under-approximation using test suite
- replace $range_l(x)$ by $range_l^+(x)$ resp. $range_l^-(x)$

$$\Rightarrow \quad \delta_\gamma^+ \leq \delta_\gamma \leq \delta_\gamma^-$$

- perfect approximation: $\delta_\gamma^+ = \delta_\gamma^-$

$\rightarrow$ exact computation of $range_l(x)$ not needed

Gap Approximation

- program P equivalent to SPDS S $\rightarrow$ exact gaps
- program P abstracted to SPDS S $\rightarrow$ approximated gaps

- more practical: approximate $range_l(x)$ $\rightarrow$ approximated gaps
- $range_l^+(x)$: over-approximation using data-flow analyzes
- $range_l^-(x)$: under-approximation using test suite
- replace $range_l(x)$ by $range_l^+(x)$ resp. $range_l^-(x)$

$$\Rightarrow \quad \delta_\gamma^+ \le \delta_\gamma \le \delta_\gamma^-$$

- perfect approximation: $\delta_\gamma^+ = \delta_\gamma^-$

$\rightarrow$ exact computation of $range_l(x)$ not needed

Gap Approximation

- program P equivalent to SPDS S $\rightarrow$ exact gaps
- program P abstracted to SPDS S $\rightarrow$ approximated gaps

- more practical: approximate $range_l(x)$ $\rightarrow$ approximated gaps
- $range_l^+(x)$: over-approximation using data-flow analyzes
- $range_l^-(x)$: under-approximation using test suite
- replace $range_l(x)$ by $range_l^+(x)$ resp. $range_l^-(x)$

$$\Rightarrow \qquad \delta_\gamma^+ \leq \delta_\gamma \leq \delta_\gamma^-$$

- perfect approximation: $\delta_\gamma^+ = \delta_\gamma^-$

$\rightarrow$ exact computation of $range_l(x)$ not needed

Gap Approximation

- program P equivalent to SPDS S $\rightarrow$ exact gaps
- program P abstracted to SPDS S $\rightarrow$ approximated gaps

- more practical: approximate $range_l(x)$ $\rightarrow$ approximated gaps
- $range_l^+(x)$: over-approximation using data-flow analyzes
- $range_l^-(x)$: under-approximation using test suite
- replace $range_l(x)$ by $range_l^+(x)$ resp. $range_l^-(x)$

$$\Rightarrow \qquad \delta_\gamma^+ \leq \delta_\gamma \leq \delta_\gamma^-$$

- perfect approximation: $\delta_\gamma^+ = \delta_\gamma^-$

$\rightarrow$ exact computation of $range_l(x)$ not needed

Gap Approximation

- program P equivalent to SPDS S $\rightarrow$ exact gaps
- program P abstracted to SPDS S $\rightarrow$ approximated gaps

- more practical: approximate $range_l(x)$ $\rightarrow$ approximated gaps
- $range_l^+(x)$: over-approximation using data-flow analyzes
- $range_l^-(x)$: under-approximation using test suite
- replace $range_l(x)$ by $range_l^+(x)$ resp. $range_l^-(x)$

$$\Rightarrow \qquad \delta_\gamma^+ \leq \delta_\gamma \leq \delta_\gamma^-$$

- perfect approximation: $\delta_\gamma^+ = \delta_\gamma^-$

$\rightarrow$ exact computation of $range_l(x)$ not needed

Gap Approximation

- program P equivalent to SPDS S $\rightarrow$ exact gaps
- program P abstracted to SPDS S $\rightarrow$ approximated gaps

- more practical: approximate $range_l(x)$ $\rightarrow$ approximated gaps
- $range_l^+(x)$: over-approximation using data-flow analyzes
- $range_l^-(x)$: under-approximation using test suite
- replace $range_l(x)$ by $range_l^+(x)$ resp. $range_l^-(x)$

$$\Rightarrow \quad \delta_\gamma^+ \leq \delta_\gamma \leq \delta_\gamma^-$$

- perfect approximation: $\delta_\gamma^+ = \delta_\gamma^-$

$\rightarrow$ exact computation of $range_l(x)$ **not** needed

Gap Approximation

- program P equivalent to SPDS S $\rightarrow$ exact gaps
- program P abstracted to SPDS S $\rightarrow$ approximated gaps

- more practical: approximate $range_l(x)$ $\rightarrow$ approximated gaps
- $range_l^+(x)$: over-approximation using data-flow analyzes
- $range_l^-(x)$: under-approximation using test suite
- replace $range_l(x)$ by $range_l^+(x)$ resp. $range_l^-(x)$

$$\Rightarrow \qquad \delta_\gamma^+ \leq \delta_\gamma \leq \delta_\gamma^-$$

- perfect approximation: $\delta_\gamma^+ = \delta_\gamma^-$

$\rightarrow$ exact computation of $range_l(x)$ **not** needed

Gap Approximation

- program P equivalent to SPDS S $\rightarrow$ exact gaps
- program P abstracted to SPDS S $\rightarrow$ approximated gaps

- more practical: approximate $range_l(x)$ $\rightarrow$ approximated gaps
- $range_l^+(x)$: over-approximation using data-flow analyzes
- $range_l^-(x)$: under-approximation using test suite
- replace $range_l(x)$ by $range_l^+(x)$ resp. $range_l^-(x)$

$$\Rightarrow \qquad \delta_\gamma^+ \leq \delta_\gamma \leq \delta_\gamma^-$$

- perfect approximation: $\delta_\gamma^+ = \delta_\gamma^-$

$\rightarrow$ exact computation of $range_l(x)$ **not** needed

Gap Approximation

- program P equivalent to SPDS S $\rightarrow$ exact gaps
- program P abstracted to SPDS S $\rightarrow$ approximated gaps

- more practical: approximate $range_l(x)$ $\rightarrow$ approximated gaps
- $range_l^+(x)$: over-approximation using data-flow analyzes
- $range_l^-(x)$: under-approximation using test suite
- replace $range_l(x)$ by $range_l^+(x)$ resp. $range_l^-(x)$

$$\Rightarrow \qquad \delta_\gamma^+ \leq \delta_\gamma \leq \delta_\gamma^-$$

- perfect approximation: $\delta_\gamma^+ = \delta_\gamma^-$

$\rightarrow$ exact computation of $range_l(x)$ **not** needed

Summary

- presented: framework to compute exact code coverage gaps
- ISO-C programs mapped to symbolic pushdown systems

→ exact range computation (not Turing powerful)

→ exact gaps

- also presented: efficient approximation without exact ranges
- in a similar way: linear code sequence and jump coverage, jj-path/path coverage, entry/exit or loop coverage

Summary

- presented: framework to compute exact code coverage gaps
- ISO-C programs mapped to symbolic pushdown systems

→ exact range computation (not Turing powerful)

→ exact gaps

- also presented: efficient approximation without exact ranges
- in a similar way: linear code sequence and jump coverage, jj-path/path coverage, entry/exit or loop coverage

Summary

- presented: framework to compute exact code coverage gaps
- ISO-C programs mapped to symbolic pushdown systems

→ exact range computation (not Turing powerful)

→ exact gaps

- also presented: efficient approximation without exact ranges
- in a similar way: linear code sequence and jump coverage, jj-path/path coverage, entry/exit or loop coverage

Summary

- presented: framework to compute exact code coverage gaps
- ISO-C programs mapped to symbolic pushdown systems

→ exact range computation (not Turing powerful)

→ exact gaps

- also presented: efficient approximation without exact ranges
- in a similar way: linear code sequence and jump coverage, jj-path/path coverage, entry/exit or loop coverage

Summary

- presented: framework to compute exact code coverage gaps
- ISO-C programs mapped to symbolic pushdown systems

→ exact range computation (not Turing powerful)

→ exact gaps

- also presented: efficient approximation without exact ranges
- in a similar way: linear code sequence and jump coverage, jj-path/path coverage, entry/exit or loop coverage

Summary

- presented: framework to compute exact code coverage gaps
- ISO-C programs mapped to symbolic pushdown systems

→ exact range computation (not Turing powerful)

→ exact gaps

- also presented: efficient approximation without exact ranges
- in a similar way: linear code sequence and jump coverage, jj-path/path coverage, entry/exit or loop coverage

Questions?