

# 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% → 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% → 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% → 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
  - 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% → 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

```

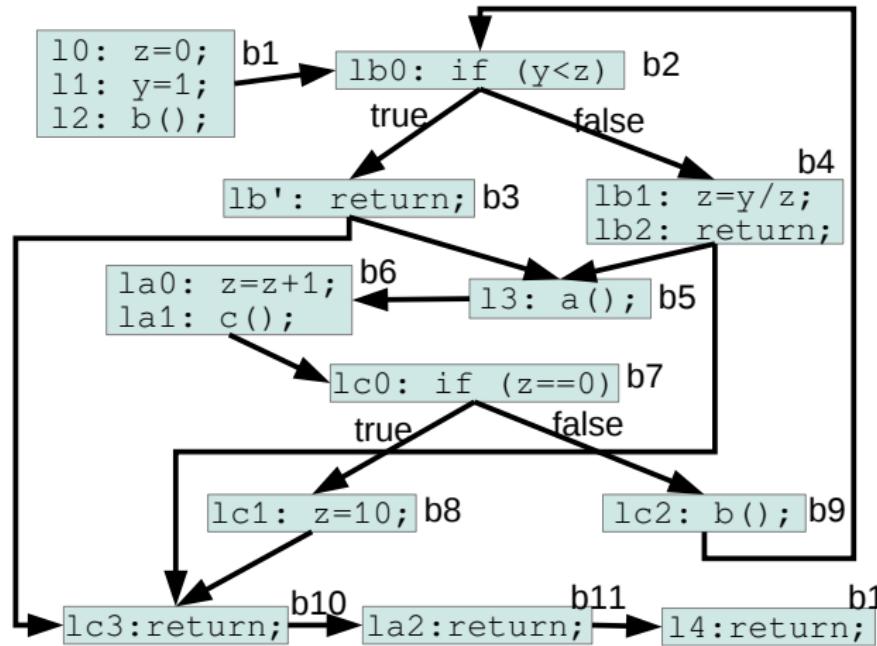
char y, z;
void b() {
    l b0: if(y<z) return;
    l b1: z=y/z;
    l b2: return; }

void c() {
    l c0: if (z == 0)
        l c1: z = 10;
    else l c2: b();
    l c3: return; }

void a() {
    l a0: z=z+1;
    l a1: c();
    l a2: 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:  $\text{blocks}(P)$ ,  $\text{edges}(P)$ : interprocedural control flow

# Code Coverage Metrics $\gamma$

- test suite  $t \in T_P$ : set of tests  $t = \{\alpha_1, \alpha_2, \dots\}$  for program  $P$
- code coverage metric  $\gamma : T_P \rightarrow [0, 1]$  monotonically increasing
- function coverage  $\gamma_f^P(t) := |\text{func}(t)| / |\text{func}(P)|$
- statement coverage  $\gamma_s^P(t) := |\text{stats}(t)| / |\text{stats}(P)|$
- decision coverage  $\gamma_d^P(t) := |\text{edges}(t)| / |\text{edges}(P)|$
- branch coverage  $\gamma_b^P(t) := |\text{blocks}(t)| / |\text{blocks}(P)|$
- $bExpr(l) = \{e_1, e_2, \dots, e_n\}$  ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \mid l \in \text{labels}(P), e \in bExpr(l)\}$
- condition or predicate coverage metric:  
 $\gamma_c^P(t) := |\text{exval}(t, P)| / (2 \cdot |BExpr(P)|)$

# Code Coverage Metrics $\gamma$

- test suite  $t \in T_P$ : set of tests  $t = \{\alpha_1, \alpha_2, \dots\}$  for program  $P$
- code coverage metric  $\gamma : T_P \rightarrow [0, 1]$  monotonically increasing
- function coverage  $\gamma_f^P(t) := |\text{func}(t)| / |\text{func}(P)|$
- statement coverage  $\gamma_s^P(t) := |\text{stats}(t)| / |\text{stats}(P)|$
- decision coverage  $\gamma_d^P(t) := |\text{edges}(t)| / |\text{edges}(P)|$
- branch coverage  $\gamma_b^P(t) := |\text{blocks}(t)| / |\text{blocks}(P)|$
- $bExpr(l) = \{e_1, e_2, \dots, e_n\}$  ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \mid l \in \text{labels}(P), e \in bExpr(l)\}$
- condition or predicate coverage metric:  
 $\gamma_c^P(t) := |\text{exval}(t, P)| / (2 \cdot |BExpr(P)|)$

# Code Coverage Metrics $\gamma$

- test suite  $t \in T_P$ : set of tests  $t = \{\alpha_1, \alpha_2, \dots\}$  for program  $P$
- code coverage metric  $\gamma : T_P \rightarrow [0, 1]$  monotonically increasing
- **function coverage**  $\gamma_f^P(t) := |\text{func}(t)| / |\text{func}(P)|$
- statement coverage  $\gamma_s^P(t) := |\text{stats}(t)| / |\text{stats}(P)|$
- decision coverage  $\gamma_d^P(t) := |\text{edges}(t)| / |\text{edges}(P)|$
- branch coverage  $\gamma_b^P(t) := |\text{blocks}(t)| / |\text{blocks}(P)|$
- $bExpr(l) = \{e_1, e_2, \dots, e_n\}$  ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \mid l \in \text{labels}(P), e \in bExpr(l)\}$
- condition or predicate coverage metric:  
 $\gamma_c^P(t) := |\text{exval}(t, P)| / (2 \cdot |BExpr(P)|)$

# Code Coverage Metrics $\gamma$

- test suite  $t \in T_P$ : set of tests  $t = \{\alpha_1, \alpha_2, \dots\}$  for program  $P$
- code coverage metric  $\gamma : T_P \rightarrow [0, 1]$  monotonically increasing
- function coverage  $\gamma_f^P(t) := |\text{func}(t)| / |\text{func}(P)|$
- statement coverage  $\gamma_s^P(t) := |\text{stats}(t)| / |\text{stats}(P)|$
- decision coverage  $\gamma_d^P(t) := |\text{edges}(t)| / |\text{edges}(P)|$
- branch coverage  $\gamma_b^P(t) := |\text{blocks}(t)| / |\text{blocks}(P)|$
- $bExpr(l) = \{e_1, e_2, \dots, e_n\}$  ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \mid l \in \text{labels}(P), e \in bExpr(l)\}$
- condition or predicate coverage metric:  
 $\gamma_c^P(t) := |\text{exval}(t, P)| / (2 \cdot |BExpr(P)|)$

# Code Coverage Metrics $\gamma$

- test suite  $t \in T_P$ : set of tests  $t = \{\alpha_1, \alpha_2, \dots\}$  for program  $P$
- code coverage metric  $\gamma : T_P \rightarrow [0, 1]$  monotonically increasing
- function coverage  $\gamma_f^P(t) := |\text{func}(t)| / |\text{func}(P)|$
- statement coverage  $\gamma_s^P(t) := |\text{stats}(t)| / |\text{stats}(P)|$
- decision coverage  $\gamma_d^P(t) := |\text{edges}(t)| / |\text{edges}(P)|$
- branch coverage  $\gamma_b^P(t) := |\text{blocks}(t)| / |\text{blocks}(P)|$
  
- $bExpr(l) = \{e_1, e_2, \dots, e_n\}$  ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \mid l \in \text{labels}(P), e \in bExpr(l)\}$
- condition or predicate coverage metric:  
$$\gamma_c^P(t) := |\text{exval}(t, P)| / (2 \cdot |BExpr(P)|)$$

# Code Coverage Metrics $\gamma$

- test suite  $t \in T_P$ : set of tests  $t = \{\alpha_1, \alpha_2, \dots\}$  for program  $P$
- code coverage metric  $\gamma : T_P \rightarrow [0, 1]$  monotonically increasing
- function coverage  $\gamma_f^P(t) := |\text{func}(t)| / |\text{func}(P)|$
- statement coverage  $\gamma_s^P(t) := |\text{stats}(t)| / |\text{stats}(P)|$
- decision coverage  $\gamma_d^P(t) := |\text{edges}(t)| / |\text{edges}(P)|$
- branch coverage  $\gamma_b^P(t) := |\text{blocks}(t)| / |\text{blocks}(P)|$
- $bExpr(l) = \{e_1, e_2, \dots, e_n\}$  ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \bullet l \in \text{labels}(P), e \in bExpr(l)\}$
- condition or predicate coverage metric:  
$$\gamma_c^P(t) := |\text{exval}(t, P)| / (2 \cdot |BExpr(P)|)$$

# Code Coverage Metrics $\gamma$

- test suite  $t \in T_P$ : set of tests  $t = \{\alpha_1, \alpha_2, \dots\}$  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 $\gamma$

- test suite  $t \in T_P$ : set of tests  $t = \{\alpha_1, \alpha_2, \dots\}$  for program  $P$
- code coverage metric  $\gamma : T_P \rightarrow [0, 1]$  monotonically increasing
- function coverage  $\gamma_f^P(t) := |\text{func}(t)| / |\text{func}(P)|$
- statement coverage  $\gamma_s^P(t) := |\text{stats}(t)| / |\text{stats}(P)|$
- decision coverage  $\gamma_d^P(t) := |\text{edges}(t)| / |\text{edges}(P)|$
- branch coverage  $\gamma_b^P(t) := |\text{blocks}(t)| / |\text{blocks}(P)|$
- $bExpr(l) = \{e_1, e_2, \dots, e_n\}$  ... set of all boolean sub-expressions
- $BExpr(P) := \{(l, e) \bullet l \in \text{labels}(P), e \in bExpr(l)\}$
- condition or predicate coverage metric:  
$$\gamma_c^P(t) := |\text{exval}(t, P)| / (2 \cdot |BExpr(P)|)$$

# Code Coverage Metrics $\gamma$

- test suite  $t \in T_P$ : set of tests  $t = \{\alpha_1, \alpha_2, \dots\}$  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 Metric Gap $\delta$

- 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  $\delta_\gamma$   
 $\rightarrow$  adequate modeling

# Code Coverage Metric Gap $\delta$

- 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  $\delta_\gamma$   
 $\rightarrow$  adequate modeling

# Code Coverage Metric Gap $\delta$

- 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  $\delta_\gamma$   
 $\rightarrow$  adequate modeling

# Code Coverage Metric Gap $\delta$

- 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 → more accurate computat. of gap  $\delta_\gamma$   
→ adequate modeling

# Code Coverage Metric Gap $\delta$

- 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 → more accurate computat. of gap  $\delta_\gamma$**   
→ adequate modeling

# Code Coverage Metric Gap $\delta$

- 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 → more accurate computat. of gap  $\delta_\gamma$
- **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  $[e_1]$
  - $f(x_1, \dots, x_n)$ ; function call (call by value)
  - $\{s\}$ ; loop body (single statement)
  - $\{s_1; \dots; s_m\}$ ; loop body (multiple statements)
- 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  $\llbracket e_2 \rrbracket$  into variable  $x$  at index  $\llbracket e_1 \rrbracket$
  - $f(x_1, \dots, x_n);$  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  $[e_1]$
  - $f(x_1, \dots, x_n)$ ; 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  $\llbracket e_2 \rrbracket$  into variable  $x$  at index  $\llbracket e_1 \rrbracket$
  - $f(x_1, \dots, x_n);$  **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  $\llbracket e_2 \rrbracket$  into variable  $x$  at index  $\llbracket e_1 \rrbracket$
  - $f(x_1, \dots, x_n);$  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  $\llbracket e_2 \rrbracket$  into variable  $x$  at index  $\llbracket e_1 \rrbracket$
  - $f(x_1, \dots, x_n);$  function call (call by value)
  - $return;$  function return
  - $\textcolor{red}{if} (e) goto l';$  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  $\llbracket e_2 \rrbracket$  into variable  $x$  at index  $\llbracket e_1 \rrbracket$
  - $f(x_1, \dots, x_n);$  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  $vdbl$ , functions  $func$ ,  $main \in func$
- variable  $x$  (array) has integer type  $bits(x)$  and length  $\text{len}(x)$
- 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 /
- $g : vdbl \times \mathbb{Z} \rightarrow \mathbb{Z}$ ,  $c : vdbl(l) \times \mathbb{Z} \rightarrow \mathbb{Z}$  ...variable settings
- $g(x, i), c(x, i)$  current value of variable  $x$  on index  $i$

# Symbolic Pushdown System (SPDS)

- SPDS  $S$ : global variables  $vdbl$ , functions  $func$ ,  $main \in func$
- variable  $x$  (array) has integer type  $bits(x)$  and length  $\text{len}(x)$
- 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  $l$
- $g : vdbl \times \mathbb{Z} \rightarrow \mathbb{Z}$ ,  $c : vdbl(l) \times \mathbb{Z} \rightarrow \mathbb{Z}$  ...variable settings
- $g(x, i), c(x, i)$  current value of variable  $x$  on index  $i$

# Symbolic Pushdown System (SPDS)

- SPDS  $S$ : global variables  $vdbl$ , functions  $func$ ,  $main \in func$
- variable  $x$  (array) has integer type  $bits(x)$  and length  $\text{len}(x)$
- 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 /
- $g : vdbl \times \mathbb{Z} \rightarrow \mathbb{Z}$ ,  $c : vdbl(l) \times \mathbb{Z} \rightarrow \mathbb{Z}$  ...variable settings
- $g(x, i), c(x, i)$  current value of variable  $x$  on index  $i$

# Symbolic Pushdown System (SPDS)

- SPDS  $S$ : global variables  $vgbI$ , 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 /
- $g : vgbI \times \mathbb{Z} \rightarrow \mathbb{Z}$ ,  $c : vlcI(l) \times \mathbb{Z} \rightarrow \mathbb{Z}$  ...variable settings
- $g(x, i), c(x, i)$  current value of variable  $x$  on index  $i$

# Symbolic Pushdown System (SPDS)

- SPDS  $S$ : global variables  $vgl$ , functions  $func$ ,  $main \in func$
- variable  $x$  (array) has integer type  $bits(x)$  and length  $\text{len}(x)$
- 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  $l$
- $g : vgl \times \mathbb{Z} \rightarrow \mathbb{Z}$ ,  $c : vlc(l) \times \mathbb{Z} \rightarrow \mathbb{Z}$  ...variable settings
- $g(x, i), c(x, i)$  current value of variable  $x$  on index  $i$

# Symbolic Pushdown System (SPDS)

- SPDS  $S$ : global variables  $vgl$ , functions  $func$ ,  $main \in func$
- variable  $x$  (array) has integer type  $bits(x)$  and length  $\text{len}(x)$
- 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  $l$
- $g : vgl \times \mathbb{Z} \rightarrow \mathbb{Z}$ ,  $c : vcl(l) \times \mathbb{Z} \rightarrow \mathbb{Z}$  ...variable settings
- $g(x, i), c(x, i)$  current value of variable  $x$  on index  $i$

# Symbolic Pushdown System (SPDS)

- SPDS  $S$ : global variables  $vgbI$ , 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  $l$
- $g : vgbI \times \mathbb{Z} \rightarrow \mathbb{Z}$ ,  $c : vlcI(l) \times \mathbb{Z} \rightarrow \mathbb{Z}$  ...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  $\gamma : T_P \rightarrow [0, 1]$

- 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  $\delta_\gamma(S)$
- 5 Conclude size of the gap  $\delta_\gamma(P)$

# 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  $\delta_\gamma(S)$
- 5 Conclude size of the gap  $\delta_\gamma(P)$

# 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  $\delta_\gamma(S)$
- 5 Conclude size of the gap  $\delta_\gamma(P)$

# 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  $\delta_\gamma(S)$
- 5 Conclude size of the gap  $\delta_\gamma(P)$

# 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  $\delta_\gamma(S)$
- 5 Conclude size of the gap  $\delta_\gamma(P)$

# 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)

→ ISO-C statements are mapped to basic statements (abbreviations) in the SPDS model.

→ Higher level concepts can be simulated by combining basic statements.

→ The SPDS model is a state transition system with states and transitions.

→ The states represent the memory state of the program.

→ The transitions represent the execution of statements.

→ The transitions are labeled with the abbreviations of the ISO-C statements.

→ The transitions are labeled with the abbreviations of the ISO-C statements.

→ The transitions are labeled with the abbreviations of the ISO-C statements.

→ The transitions are labeled with the abbreviations of the ISO-C statements.

- 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
  - function calls are simulated via stack frames
  - loops are simulated via stack frames
  - function calls are simulated via stack frames
  - loops are simulated via stack frames
  - 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 *iretf* 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 *iretf* 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 *iretf* 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 *iretf*, 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<sub>f</sub>* 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  $i_{ret_f}$  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  $i_{ret_f}$  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 3 - Extraction of Exact Variable Ranges

- model checker Moped: algorithm to create  $\text{Post}^*$  automaton

- $\text{Post}^*$  accepts reachable configurations (infinite set)

- map SPDS  $S$  to Remopla (input language of Moped)

→ exact variable range $_{\mathcal{I}}(x) = \{\llbracket x \rrbracket_g^c \bullet (g, [(l, c)...]) \in \text{Post}^*(S)\}$

- problem:  $\text{Post}^*(S)$  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  $range_l(x) = \{[\![x]\!]_g^c \mid (g, [(l, c)...]) \in Post^*(S)\}$
- 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  $range_l(x) = \{[\![x]\!]_g^c \mid (g, [(l, c)\dots]) \in Post^*(S)\}$

- 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 $_I(x) = \{[\![x]\!]_g^c \bullet (g, [(I, c)...]) \in Post^*(S)\}$
- 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  $range_l(x) = \{\llbracket x \rrbracket_g^c \bullet (g, [(l, c)...]) \in Post^*(S)\}$
- problem:  $Post^*(S)$  infinite

## Step 3 - problem: $\text{Post}^*(S)$ infinite

- solution: restrict on reachable heads  $h(S) := \{(g, (l, c)) \dots\}$   
→ finite, because of finite variable types
- implementation: symbolical computation of...

Implementation of the computation of the set of reachable heads  $h(S)$  for a given set of terms  $S$ .

The computation is based on a breadth-first search (BFS) of the term graph.

The search starts at the root node  $S$  and explores all possible transitions to new nodes.

The search continues until no new nodes can be reached, indicating that the search space is finite.

The resulting set of nodes is the set of reachable heads  $h(S)$ .

## Step 3 - problem: $\text{Post}^*(S)$ infinite

- solution: restrict on reachable heads  $h(S) := \{(g, (l, c)) \dots\}$   
→ 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  $\text{Post}^*$
  2. characteristic function  $r: \{0,1\}^m \rightarrow \{0,1\}$  for  $\text{range}(x)$  using OBDD complementation out of  $\text{Post}^*$

## Step 3 - problem: $\text{Post}^*(S)$ infinite

- solution: restrict on reachable heads  $h(S) := \{(g, (l, c)) \dots\}$   
→ 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  $\text{Post}^*$
  - 2 characteristic function  $r : \{0, 1\}^m \rightarrow \{0, 1\}$  for  $\text{range}_l(x)$  using OBDD co-factorization out of  $h(S)$

## Step 3 - problem: $\text{Post}^*(S)$ infinite

- solution: restrict on reachable heads  $h(S) := \{(g, (l, c)) \dots\}$   
→ 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  $\text{Post}^*$
  - 2 characteristic function  $r : \{0,1\}^m \rightarrow \{0,1\}$  for  $\text{range}_l(x)$  using OBDD co-factorization out of  $h(S)$

## Step 3 - problem: $\text{Post}^*(S)$ infinite

- solution: restrict on reachable heads  $h(S) := \{(g, (l, c)) \dots\}$   
→ 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  $\text{Post}^*$
  - 2 characteristic function  $r : \{0,1\}^m \rightarrow \{0,1\}$  for  $\text{range}_l(x)$  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  $x \notin vgbI(S) \rightsquigarrow SPDS S'$
- $bits(x) = 1, len(x) = 1$
- observation:  $x$  has random value  $\llbracket x \rrbracket \in \{0, 1\}$  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  $S$

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

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

$$\Rightarrow \delta_B(S) = 1 - \frac{|\{b \in blocks(S) \mid range_{bit(b)}^{S'}(x) \neq \emptyset\}|}{|blocks(S)|}$$

# 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  $x \notin vgbI(S) \rightsquigarrow SPDS S'$
- $\text{bits}(x) = 1, \text{len}(x) = 1$
- observation:  $x$  has random value  $\llbracket x \rrbracket \in \{0, 1\}$  on each **reachable** label (function entry/block entry)
- compute exact  $\text{range}_l(x) \quad \forall l \in \text{labels}(S')$
- $\text{range}_l(x) \neq \emptyset \Leftrightarrow$  label  $l$  reachable in  $S$

$$\Rightarrow \delta_s(S) = 1 - \frac{|\{l \in \text{stats}(S) \mid \text{range}_{\text{bit}(l)}^{S'}(x) \neq \emptyset\}|}{|\text{stats}(S)|}$$

$$\Rightarrow \delta_f(S) = 1 - \frac{|\{f \in \text{func}(S) \mid \text{range}_{\text{bit}(f)}^{S'}(x) \neq \emptyset\}|}{|\text{func}(S)|}$$

$$\Rightarrow \delta_b(S) = 1 - \frac{|\{b \in \text{blocks}(S) \mid \text{range}_{\text{bit}(b)}^{S'}(x) \neq \emptyset\}|}{|\text{blocks}(S)|}$$

# 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  $x \notin vgbI(S) \rightsquigarrow SPDS S'$
- $bits(x) = 1$ ,  $len(x) = 1$
- observation:**  $x$  has random value  $\llbracket x \rrbracket \in \{0, 1\}$  on each **reachable** label (function entry/block entry)
- compute exact  $range_I(x) \quad \forall I \in labels(S')$
- $range_I(x) \neq \emptyset \Leftrightarrow$  label  $I$  reachable in  $S$

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

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

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

# 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  $x \notin vgbI(S) \rightsquigarrow SPDS S'$
- $bits(x) = 1$ ,  $len(x) = 1$
- observation:  $x$  has random value  $\llbracket x \rrbracket \in \{0, 1\}$  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  $S$

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

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

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

# 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  $x \notin vgbI(S) \rightsquigarrow SPDS S'$
- $bits(x) = 1$ ,  $len(x) = 1$
- observation:  $x$  has random value  $\llbracket x \rrbracket \in \{0, 1\}$  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 / 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  $x \notin vgbI(S) \rightsquigarrow SPDS S'$
- $bits(x) = 1$ ,  $len(x) = 1$
- observation:  $x$  has random value  $\llbracket x \rrbracket \in \{0, 1\}$  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  $x \notin vgbI(S) \rightsquigarrow SPDS S'$
- $bits(x) = 1$ ,  $len(x) = 1$
- observation:  $x$  has random value  $\llbracket x \rrbracket \in \{0, 1\}$  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  $x \notin vgbI(S) \rightsquigarrow SPDS S'$
- $bits(x) = 1$ ,  $len(x) = 1$
- observation:  $x$  has random value  $\llbracket x \rrbracket \in \{0, 1\}$  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  $bits(x) = len(x) = 1$

l<sub>2</sub>: call function  $b$

l<sub>b0</sub>:  $(y < z) \rightsquigarrow false$

l<sub>b1</sub>: division-by-zero

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

$$\Rightarrow \delta_s(S) = 1 - \frac{|\{l \in \text{state}(S) \mid range_f^{S'}(x) \neq \emptyset\}|}{|\text{state}(S)|} = 67\%$$

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

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

- 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  $bits(x) = len(x) = 1$

- I2: call function  $b$**

- l<sub>b0</sub>:  $(y < z) \rightsquigarrow false$

- l<sub>b1</sub>: division-by-zero

$$\Rightarrow range_l^{S'}(x) = \emptyset \Leftrightarrow l \notin \{l_0, l_1, l_2, l_{b0}, l_{b1}\}$$

$$\Rightarrow \delta_s(S) = 1 - \frac{|\{l \in \text{state}(S) \mid range_l^{S'}(x) \neq \emptyset\}|}{|\text{state}(S)|} = 67\%$$

$$\Rightarrow \delta_f(S) = 1 - \frac{|\{l \in \text{func}(S) \mid range_{fct(l)}^{S'}(x) \neq \emptyset\}|}{|\text{func}(S)|} = 50\%$$

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

- 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  $bits(x) = len(x) = 1$
- I2: call function  $b$
- $l_{b0}$ :  $(y < z) \rightsquigarrow false$

l<sub>b1</sub>: division-by-zero  
 $\rightarrow range_l^{S'}(x) = \emptyset \Leftrightarrow l \notin \{l_0, l_1, l_2, l_{b0}, l_{b1}\}$

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

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

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

- 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  $bits(x) = len(x) = 1$

- l<sub>2</sub>: call function  $b$

- l<sub>b0</sub>:  $(y < z) \rightsquigarrow false$

- l<sub>b1</sub>: division-by-zero

$$\rightarrow range_l^{S'}(x) = \emptyset \Leftrightarrow l \notin \{l_0, l_1, l_2, l_{b0}, l_{b1}\}$$

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

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

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

- 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  $bits(x) = len(x) = 1$
  - $I/2$ : call function  $b$
  - $l/b0$ :  $(y < z) \rightsquigarrow false$
  - $l/b1$ : division-by-zero
- $\rightarrow range_I^{S'}(x) = \emptyset \Leftrightarrow I \notin \{l0, l1, l2, l/b0, l/b1\}$

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

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

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

- 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  $bits(x) = len(x) = 1$

- I2: call function  $b$

- l<sub>b0</sub>:  $(y < z) \rightsquigarrow false$

- l<sub>b1</sub>: division-by-zero

$$\Rightarrow range_l^{S'}(x) = \emptyset \Leftrightarrow l \notin \{l_0, l_1, l_2, l_{b0}, l_{b1}\}$$

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

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

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

- 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  $bits(x) = len(x) = 1$

- l<sub>2</sub>: call function  $b$

- l<sub>b0</sub>:  $(y < z) \rightsquigarrow false$

- l<sub>b1</sub>: division-by-zero

$$\Rightarrow range_l^{S'}(x) = \emptyset \Leftrightarrow l \notin \{l_0, l_1, l_2, l_{b0}, l_{b1}\}$$

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

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

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

- 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  $bits(x) = len(x) = 1$

- l<sub>2</sub>: call function  $b$

- l<sub>b0</sub>:  $(y < z) \rightsquigarrow false$

- l<sub>b1</sub>: division-by-zero

$$\Rightarrow range_l^{S'}(x) = \emptyset \Leftrightarrow l \notin \{l_0, l_1, l_2, l_{b0}, l_{b1}\}$$

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

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

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

- 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  $bits(x) = len(x) = 1$

- l<sub>2</sub>: call function  $b$

- l<sub>b0</sub>:  $(y < z) \rightsquigarrow false$

- l<sub>b1</sub>: division-by-zero

$$\Rightarrow range_l^{S'}(x) = \emptyset \Leftrightarrow l \notin \{l_0, l_1, l_2, l_{b0}, l_{b1}\}$$

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

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

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

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

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

# Step 2+4 - SPDS Supplementation and Exact Gap Inference

## Decision coverage gap $\delta_d(S)$

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

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

# Step 2+4 - SPDS Supplementation and Exact Gap Inference

## Decision coverage gap $\delta_d(S)$

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

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

## Step 2+4 - SPDS Supplementation and Exact Gap Inference

Decision coverage gap  $\delta_d(S)$ 

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

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

# Step 2+4 - SPDS Supplementation and Exact Gap Inference

## Decision coverage gap $\delta_d(S)$

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

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

# Step 2+4 - SPDS Supplementation and Exact Gap Inference

## Decision coverage gap $\delta_d(S)$

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

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

# Step 2+4 - SPDS Supplementation and Exact Gap Inference

## Decision coverage gap $\delta_d(S)$

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

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

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

```

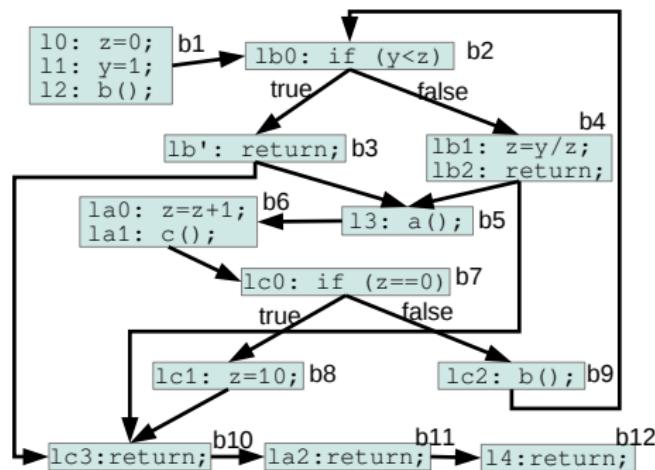
char y,z;
void b() {
    lbo: if(y<z) return
    lb1: z=y/z;
    lb2: return; }

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

```

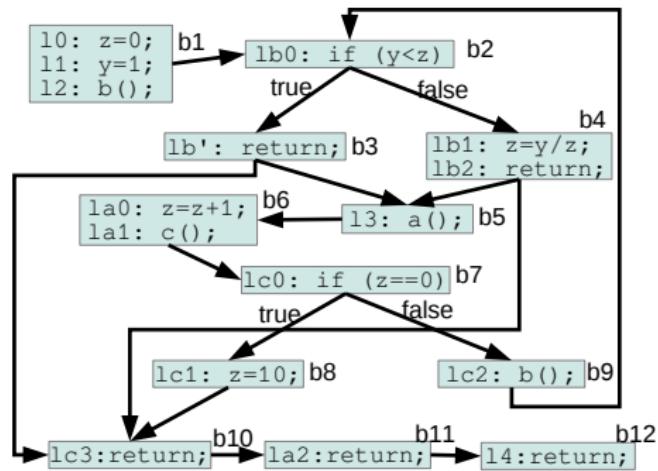
char y,z; int x(4);
void b() {
    lbo: x=2; if(y<z){x=3; return;}
    lb1: x=4; z=y/z;
    lb2: return; }

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  $range_l^{S'}(x)$  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

```

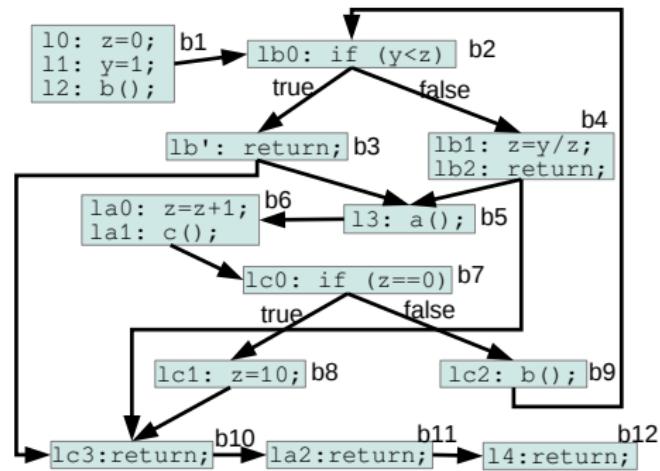
char y,z; int x(4);
void b() {
    lbo: x=2; if(y<z){x=3; return;}
    lb1: x=4; z=y/z;
    lb2: return; }

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  $range^S(x)$  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

```

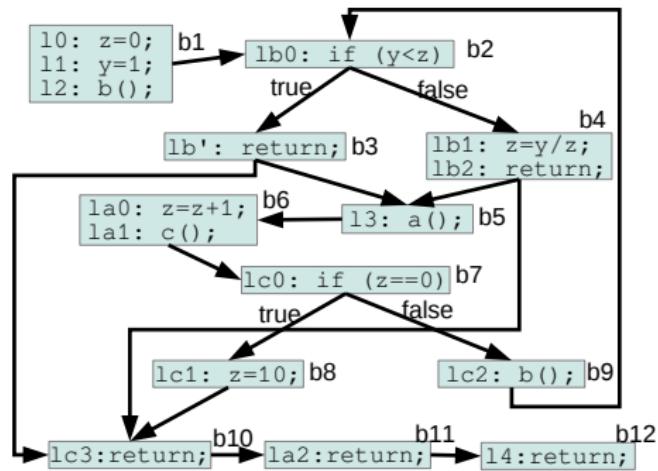
char y,z; int x(4);
void b() {
    lbo: x=2; if(y<z){x=3; return;}
    lb1: x=4; z=y/z;
    lb2: return; }

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  $range_{S'}^-(x)$  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

```

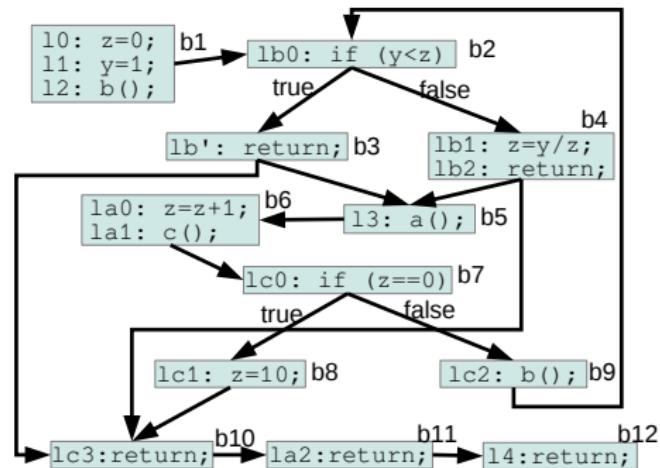
char y,z; int x(4);
void b() {
    lbo: x=2; if(y<z){x=3; return;}
    lb1: x=4; z=y/z;
    lb2: return; }

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  $range_{I'}^S(x)$  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  $x_e \notin vgbI(S) \rightsquigarrow$  SPDS  $S'$
- $\text{len}(x_e) = 1, \text{bits}(x_e) = 1$
- " $I : s$ "  $\Rightarrow$  " $I : x_{e_1} = e_1; x_{e_2} = e_2; \dots x_{e_n} = e_n; I' : s;$ "
- evaluations of  $x_{e_i}$  equivalent to covered conditions  $e_i$

$$\Rightarrow \delta_c(S) = 1 - \frac{\sum_{\substack{i \in labels(S) \\ e \in bExpr(I)}} |\text{range}_{S'}^{S'}(x_e)|}{2 \cdot |BExpr(S)|}$$

## 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  $x_e \notin vgbI(S) \rightsquigarrow$  SPDS  $S'$
- $\text{len}(x_e) = 1, \text{bits}(x_e) = 1$
- " $I : s$ "  $\Rightarrow$  " $I : x_{e_1} = e_1; x_{e_2} = e_2; \dots x_{e_n} = e_n; I' : s;$ "
- evaluations of  $x_{e_i}$  equivalent to covered conditions  $e_i$

$$\Rightarrow \delta_c(S) = 1 - \frac{\sum_{\substack{l \in labels(S) \\ \text{such that } bExpr(l)}} |\text{range}_{\gamma'}^{S'}(x_e)|}{2 \cdot |BExpr(S)|}$$

# 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  $x_e \notin vgbI(S) \rightsquigarrow$  SPDS  $S'$
- $\text{len}(x_e) = 1, \text{bits}(x_e) = 1$
- " $l : s$ "  $\Rightarrow$  " $l : x_{e_1} = e_1; x_{e_2} = e_2; \dots x_{e_n} = e_n; l' : s;$ "
- evaluations of  $x_{e_i}$  equivalent to covered conditions  $e_i$

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

## 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  $x_e \notin vgbI(S) \rightsquigarrow$  SPDS  $S'$
- $\text{len}(x_e) = 1, \text{bits}(x_e) = 1$
- " $I : s$ "  $\Rightarrow$  " $I : x_{e_1} = e_1; x_{e_2} = e_2; \dots x_{e_n} = e_n; I' : s;$ "
- evaluations of  $x_{e_i}$  equivalent to covered conditions  $e_i$

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

# 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  $x_e \notin vgbI(S) \rightsquigarrow$  SPDS  $S'$
- $\text{len}(x_e) = 1, \text{bits}(x_e) = 1$
- " $l : s$ "  $\Rightarrow$  " $l : x_{e_1} = e_1; x_{e_2} = e_2; \dots x_{e_n} = e_n; l' : s;$ "
- evaluations of  $x_e$ ; equivalent to covered conditions  $e$ ;

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

## 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  $x_e \notin vgbI(S) \rightsquigarrow$  SPDS  $S'$
- $\text{len}(x_e) = 1, \text{bits}(x_e) = 1$
- " $I : s$ "  $\Rightarrow$  " $I : x_{e_1} = e_1; x_{e_2} = e_2; \dots x_{e_n} = e_n; I' : s;$ "
- evaluations of  $x_{e_i}$  equivalent to covered conditions  $e_i$

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

# 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  $range_l^{S'}(x) \subseteq \{true, false\}$
- $range_p^{S'}(x) = \{false\}$  indicates  $y < z$  evals.
- $range_q^{S'}(x) = \emptyset$  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  $range_l^{S'}(x) \subseteq \{true, false\}$
- $range_p^{S'}(x) = \{false\}$  indicates  $y < z$  evals.
- $range_q^{S'}(x) = \emptyset$  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  $range_l^{S'}(x) \subseteq \{true, false\}$
  - $range_p^{S'}(x) = \{false\}$  indicates  $y < z$  evals.
  - $range_q^{S'}(x) = \emptyset$  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() {
    |b0: x=(y<z); p: if (y<z) return;
    |b1: z=y/z;
    |b2: return; }

void c() {
    |c0: x=(z==0); q: if (z == 0)
        |c1: z = 10;
    else |c2: b();
    |c3: return; }

void a() {
    |a0: z=z+1;
    |a1: c();
    |a2: return; }

void main() {
    |0: z=0;
    |1: y=1;
    |2: b();
    |3: a();
    |4: return; }

```

- exact  $range_I^{S'}(x) \subseteq \{true, false\}$
- $range_p^{S'}(x) = \{false\}$  indicates  $y < z$  evals.
- $range_q^{S'}(x) = \emptyset$  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() {
    |b0: x=(y<z); p: if (y<z) return;
    |b1: z=y/z;
    |b2: return; }

void c() {
    |c0: x=(z==0); q: if (z == 0)
        |c1: z = 10;
    else |c2: b();
    |c3: return; }

void a() {
    |a0: z=z+1;
    |a1: c();
    |a2: return; }

void main() {
    |0: z=0;
    |1: y=1;
    |2: b();
    |3: a();
    |4: return; }

```

- exact  $range_I^{S'}(x) \subseteq \{true, false\}$
- $range_p^{S'}(x) = \{false\}$  indicates  $y < z$  evals.
- $range_q^{S'}(x) = \emptyset$  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  $\text{range}_I(x) \rightarrow$  approximated gaps
- $\text{range}_I^+(x)$ : over-approximation using data-flow analyzes
- $\text{range}_I^-(x)$ : under-approximation using test suite
- replace  $\text{range}_I(x)$  by  $\text{range}_I^+(x)$  resp.  $\text{range}_I^-(x)$

$$\Rightarrow \quad \delta_I^+ \leq \delta_I \leq \delta_I^-$$

- perfect approximation:  $\delta_I^+ = \delta_I^-$
- exact computation of  $\text{range}_I(x)$  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  $\text{range}_I(x) \rightarrow$  approximated gaps
  - $\text{range}_I^+(x)$ : over-approximation using data-flow analyzes
  - $\text{range}_I^-(x)$ : under-approximation using test suite
  - replace  $\text{range}_I(x)$  by  $\text{range}_I^+(x)$  resp.  $\text{range}_I^-(x)$

$$\Rightarrow \quad \delta_I^+ \leq \delta_I \leq \delta_I^-$$

- perfect approximation:  $\delta_I^+ = \delta_I^-$   
 $\rightarrow$  exact computation of  $\text{range}_I(x)$  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  $\text{range}_I(x) \rightarrow$  approximated gaps
  - $\text{range}_I^+(x)$ : over-approximation using data-flow analyzes
  - $\text{range}_I^-(x)$ : under-approximation using test suite
  - replace  $\text{range}_I(x)$  by  $\text{range}_I^+(x)$  resp.  $\text{range}_I^-(x)$

$$\Rightarrow \quad \delta_I^+ \leq \delta_I \leq \delta_I^-$$

- perfect approximation:  $\delta_I^+ = \delta_I^-$   
→ exact computation of  $\text{range}_I(x)$  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  $\text{range}_I(x) \rightarrow$  approximated gaps
- $\text{range}_I^+(x)$ : over-approximation using data-flow analyzes
- $\text{range}_I^-(x)$ : under-approximation using test suite
- replace  $\text{range}_I(x)$  by  $\text{range}_I^+(x)$  resp.  $\text{range}_I^-(x)$

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

- perfect approximation:  $\delta_\gamma^+ = \delta_\gamma^-$
- exact computation of  $\text{range}_I(x)$  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  $\text{range}_I(x) \rightarrow$  approximated gaps
- $\text{range}_I^+(x)$ : over-approximation using data-flow analyzes
- $\text{range}_I^-(x)$ : under-approximation using test suite
- replace  $\text{range}_I(x)$  by  $\text{range}_I^+(x)$  resp.  $\text{range}_I^-(x)$

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

- perfect approximation:  $\delta_\gamma^+ = \delta_\gamma^-$   
→ exact computation of  $\text{range}_I(x)$  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  $\text{range}_I(x) \rightarrow$  approximated gaps
- $\text{range}_I^+(x)$ : over-approximation using data-flow analyzes
- $\text{range}_I^-(x)$ : under-approximation using test suite
- replace  $\text{range}_I(x)$  by  $\text{range}_I^+(x)$  resp.  $\text{range}_I^-(x)$

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

- perfect approximation:  $\delta_\gamma^+ = \delta_\gamma^-$   
→ exact computation of  $\text{range}_I(x)$  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  $\text{range}_I(x) \rightarrow$  approximated gaps
- $\text{range}_I^+(x)$ : over-approximation using data-flow analyzes
- $\text{range}_I^-(x)$ : under-approximation using test suite
- replace  $\text{range}_I(x)$  by  $\text{range}_I^+(x)$  resp.  $\text{range}_I^-(x)$

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

- perfect approximation:  $\delta_\gamma^+ = \delta_\gamma^-$   
→ exact computation of  $\text{range}_I(x)$  **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  $\text{range}_I(x) \rightarrow$  approximated gaps
- $\text{range}_I^+(x)$ : over-approximation using data-flow analyzes
- $\text{range}_I^-(x)$ : under-approximation using test suite
- replace  $\text{range}_I(x)$  by  $\text{range}_I^+(x)$  resp.  $\text{range}_I^-(x)$

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

- perfect approximation:  $\delta_\gamma^+ = \delta_\gamma^-$   
→ exact computation of  $\text{range}_I(x)$  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  $\text{range}_I(x) \rightarrow$  approximated gaps
- $\text{range}_I^+(x)$ : over-approximation using data-flow analyzes
- $\text{range}_I^-(x)$ : under-approximation using test suite
- replace  $\text{range}_I(x)$  by  $\text{range}_I^+(x)$  resp.  $\text{range}_I^-(x)$

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

- perfect approximation:  $\delta_\gamma^+ = \delta_\gamma^-$
- exact computation of  $\text{range}_I(x)$  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?