A study of the DASH algorithm for software property checking

Kasper Føns

20083881

May 30, 2014

Table of contents

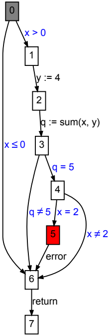
DASH$_{call}$ goal

```
void test(int x, int y)
{
  if(x > 0)
  {
    y = 4;
    int q = sum(x, y);
    if(q == 5)
      if(x == 2)
        error;
  }
}


int sum(int i, int x)
{
  int s = i + x;
  return s;
}
```

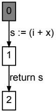
DASH$_{call}$ goal

```
void test(int x, int y)
{
  if(x > 0)
  {
    y = 4;
    int q = sum(x, y);
    if(q == 5)
      if(x == 2)
        error;
  }
}
```

$\longrightarrow$



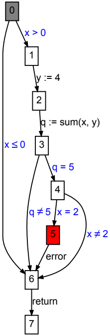

```
int sum(int i, int x)
{
  int s = i + x;
  return s;
}
```

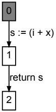


DASH$_{call}$ goal

```
void test(int x, int y)
{
  if(x > 0)
  {
    y = 4;
    int q = sum(x, y);
    if(q == 5)
      if(x == 2)
        error;
  }
}

int sum(int i, int x)
{
  int s = i + x;
  return s;
}
```

$\longrightarrow$



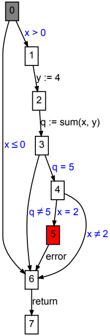


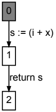
$\longrightarrow$ DASH$_{call}$

DASH$_{call}$ goal

```
void test(int x, int y)
{
  if(x > 0)
  {
    y = 4;
    int q = sum(x, y);
    if(q == 5)
      if(x == 2)
        error;
  }
}
```

```
int sum(int i, int x)
{
  int s = i + x;
  return s;
}
```



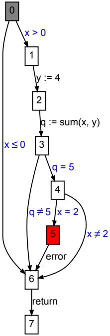


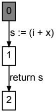
$\longrightarrow$ DASH$_{call}$ $\nearrow$ error $\searrow$ pass

DASH$_{call}$ goal

```
void test(int x, int y)
{
  if(x > 0)
  {
    y = 4;
    int q = sum(x, y);
    if(q == 5)
      if(x == 2)
        error;
  }
}

int sum(int i, int x)
{
  int s = i + x;
  return s;
}
```

⟶





⟶ DASH$_{call}$

↗ error

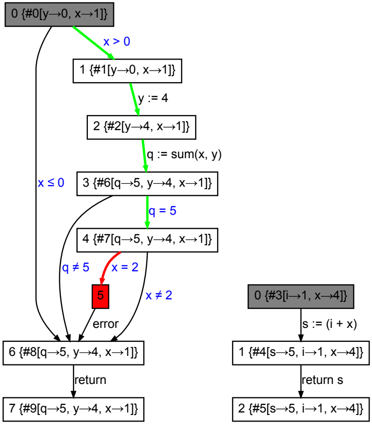
↘ **pass**

Changes needed to DASH$_{int}$

- RunTest
  Concrete execution
- ExecuteSymbolic
  Symbolic execution of traces
  - ConvertToRegionTrace...
    Generation of traces
- ExtendFrontier
  Procedure calls at the frontier

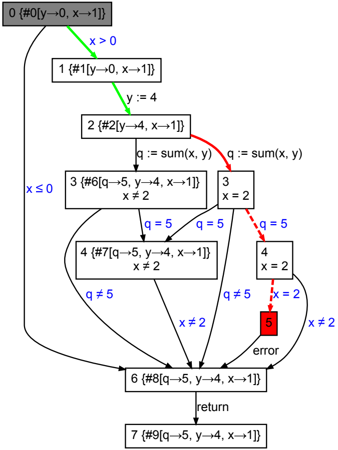
Changes needed to DASH$_{int}$

- RunTest
  Concrete execution
- **ExecuteSymbolic**
  Symbolic execution of traces
  - ConvertToRegionTrace...
    Generation of traces
- ExtendFrontier
  Procedure calls at the frontier

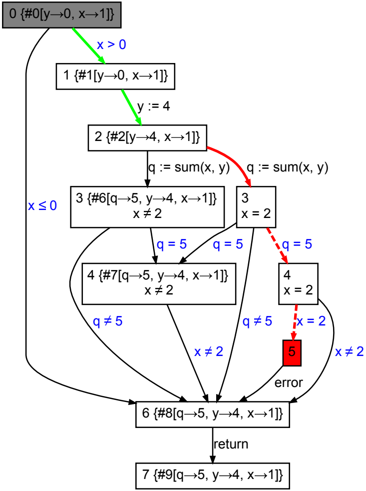


Changes needed to DASH$_{int}$

- RunTest
  Concrete execution
- ExecuteSymbolic
  Symbolic execution of traces
  - ConvertToRegionTrace...
    Generation of traces
- **ExtendFrontier**
  Procedure calls at the frontier

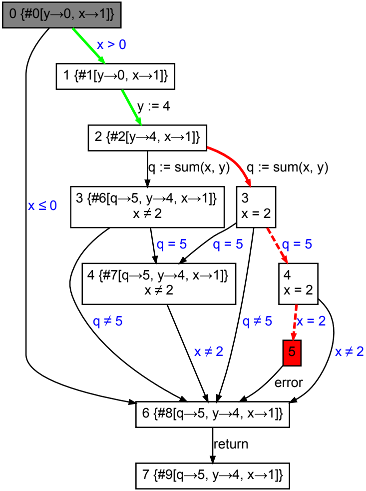


Extending beyond the frontier



- Question: Can SUM generate output q when in some state specified by w?

Extending beyond the frontier



- Question: Can SUM generate output q when in some state specified by w?
- It is a reachability question for which DASH was designed for.

Extending beyond the frontier

$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
    let $\langle \Sigma, \sigma^I, \rightarrow \rangle =$ GetProc(Edge(S_{k-1}, S_k))
    $\phi :=$ InputContraints$(\mathcal{S})$
    $\phi' := S_k[e/x]$
    $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg\phi')$
    if $r =$ FAIL **then**
        $t := m$
        $\rho :=$ true
    else
        $\rho :=$ ComputeRefinePred(m)
        $t :=$ UNSAT
    end if
else
    $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
    if $t =$ UNSAT **then**
        $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
    else
        $\rho :=$ true
    end if
end if
return $\langle t, \rho \rangle$

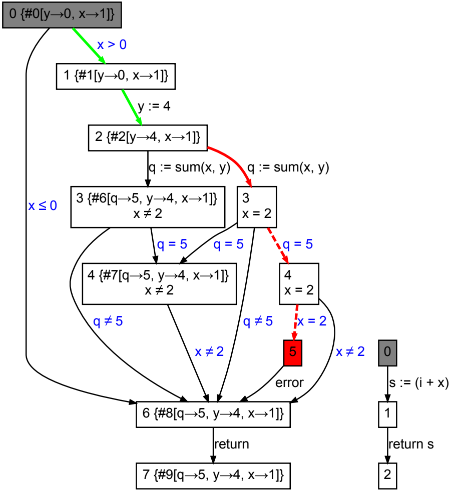
- Question: Can SUM generate output q when in some state specified by w?
- It is a reachability question for which DASH was designed for.
- ExtendFrontier modifies the graph of the sub procedure and uses DASH to answer its question.

ExtendFrontier – The Idea

$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
  let $\langle \Sigma, \sigma^I, \rightarrow \rangle =$ GetProc(Edge(S_{k-1}, S_k))
  $\phi :=$ InputContraints$(\mathcal{S})$
  $\phi' := S_k[e/x]$
  $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg\phi')$
  if $r = \text{FAIL}$ **then**
    $t := m$
    $\rho :=$ true
  else
    $\rho :=$ ComputeRefinePred(m)
    $t := \text{UNSAT}$
  end if
else
  $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
  if $t = \text{UNSAT}$ **then**
    $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
  else
    $\rho :=$ true
  end if
end if
return $\langle t, \rho \rangle$

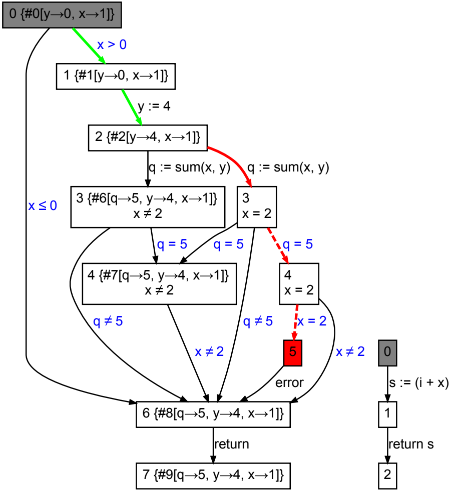
ExtendFrontier – The Idea

$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
    let $\langle \Sigma, \sigma^I, \to \rangle =$ GetProc(Edge(S_{k-1}, S_k))
    $\phi :=$ InputContraints$(\mathcal{S})$
    $\phi' := S_k[e/x]$
    $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \to \rangle, \neg \phi')$
    if $r =$ FAIL **then**
        $t := m$
        $\rho :=$ true
    else
        $\rho :=$ ComputeRefinePred(m)
        $t :=$ UNSAT
    end if
else
    $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
    if $t =$ UNSAT **then**
        $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
    else
        $\rho :=$ true
    end if
end if
return $\langle t, \rho \rangle$



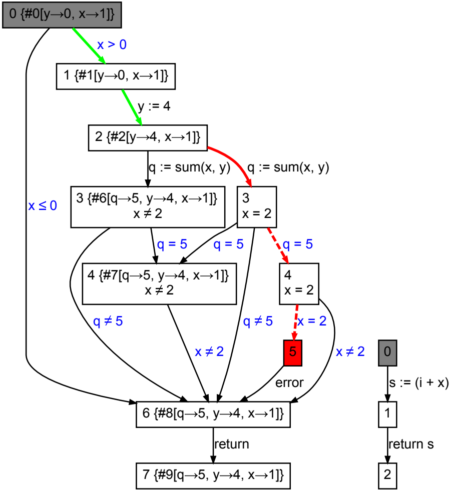
ExtendFrontier – The Idea

$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
    let $\langle \Sigma, \sigma^I, \rightarrow \rangle =$ GetProc(Edge(S_{k-1}, S_k))
    $\phi :=$ InputContraints$(\mathcal{S})$
    $\phi' := S_k[e/x]$
    $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')$
    if $r =$ FAIL **then**
        $t := m$
        $\rho :=$ true
    else
        $\rho :=$ ComputeRefinePred(m)
        $t :=$ UNSAT
    end if
else
    $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
    if $t =$ UNSAT **then**
        $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
    else
        $\rho :=$ true
    end if
end if
return $\langle t, \rho \rangle$



ExtendFrontier – The Idea

$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
    let $\langle \Sigma, \sigma^I, \rightarrow \rangle =$ GetProc(Edge(S_{k-1}, S_k))
    $\phi :=$ InputContraints$(\mathcal{S})$
    $\phi' := S_k[e/x]$
    $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg\phi')$
    if $r =$ FAIL **then**
        $t := m$
        $\rho :=$ true
    else
        $\rho :=$ ComputeRefinePred(m)
        $t :=$ UNSAT
    end if
else
    $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
    if $t =$ UNSAT **then**
        $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
    else
        $\rho :=$ true
    end if
end if
return $\langle t, \rho \rangle$



ExtendFrontier – The Idea

$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
  let $\langle \Sigma, \sigma^I, \rightarrow \rangle =$ GetProc(Edge(S_{k-1}, S_k))
  $\phi :=$ InputContraints$(\mathcal{S})$
  $\phi' := S_k[e/x]$
  $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg\phi')$
  if $r =$ FAIL **then**
    $t := m$
    $\rho :=$ true
  else
    $\rho :=$ ComputeRefinePred(m)
    $t :=$ UNSAT
  end if
else
  $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
  if $t =$ UNSAT **then**
    $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
  else
    $\rho :=$ true
  end if
end if
return $\langle t, \rho \rangle$

ExecuteSymbolic:

- $\phi_1 = x_0 > 0$
- $\mathcal{S} = \{x \mapsto x_0, y \mapsto 4\}$
- $\phi_2 = true$

ExtendFrontier – The Idea

$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge(S_{k-1}, S_k) $\in$ CallReturn(E) **then**
    let $\langle \Sigma, \sigma^I, \rightarrow \rangle =$ GetProc(Edge(S_{k-1}, S_k))
    $\phi :=$ InputContraints($\mathcal{S}$)
    $\phi' := S_k[e/x]$
    $\langle r, m \rangle :=$ DASH($\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg\phi'$)
    if $r = \text{FAIL}$ **then**
        $t := m$
        $\rho :=$ true
    else
        $\rho :=$ ComputeRefinePred(m)
        $t := \text{UNSAT}$
    end if
else
    $t :=$ IsSAT($\phi_1, \mathcal{S}, \phi_2, P$)
    if $t = \text{UNSAT}$ **then**
        $\rho :=$ RefinePred($\mathcal{S}, \tau_w$)
    else
        $\rho :=$ true
    end if
end if
return $\langle t, \rho \rangle$

ExecuteSymbolic:

- $\phi_1 = x_0 > 0$
- $\mathcal{S} = \{x \mapsto x_0, y \mapsto 4\}$
- $\phi_2 = true$

ExtendFrontier – The Idea

$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
  let $\langle \Sigma, \sigma^I, \to \rangle =$ GetProc(Edge(S_{k-1}, S_k))
  $\phi :=$ InputContraints$(\mathcal{S})$
  $\phi' := S_k[e/x]$
  $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \to \rangle, \neg\phi')$
  if $r =$ FAIL **then**
    $t := m$
    $\rho :=$ true
  else
    $\rho :=$ ComputeRefinePred(m)
    $t :=$ UNSAT
  end if
else
  $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
  if $t =$ UNSAT **then**
    $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
  else
    $\rho :=$ true
  end if
end if
return $\langle t, \rho \rangle$

ExecuteSymbolic:

- $\phi_1 = x_0 > 0$
- $\mathcal{S} = \{x \mapsto x_0, y \mapsto 4\}$
- $\phi_2 = true$

ExtendFrontier – The Idea

$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
  let $\langle \Sigma, \sigma^I, \rightarrow \rangle =$ GetProc(Edge(S_{k-1}, S_k))
  $\phi :=$ InputContraints$(\mathcal{S})$
  $\phi' := S_k[e/x]$
  $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg\phi')$
  if $r =$ FAIL **then**
    $t := m$
    $\rho :=$ true
  else
    $\rho :=$ ComputeRefinePred(m)
    $t :=$ UNSAT
  end if
else
  $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
  if $t =$ UNSAT **then**
    $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
  else
    $\rho :=$ true
  end if
end if
return $\langle t, \rho \rangle$

ExecuteSymbolic:

- $\phi_1 = x_0 > 0$
- $\mathcal{S} = \{x \mapsto x_0, y \mapsto 4\}$
- $\phi_2 = true$

Input and exit constraints:

- $\phi = \phi_{ic}$

ExtendFrontier – The Idea

$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
  let $\langle \Sigma, \sigma^I, \rightarrow \rangle =$ GetProc(Edge(S_{k-1}, S_k))
  $\phi :=$ InputContraints$(\mathcal{S})$
  $\phi' := S_k[e/x]$
  $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg\phi')$
  if $r =$ FAIL **then**
    $t := m$
    $\rho :=$ true
  else
    $\rho :=$ ComputeRefinePred(m)
    $t :=$ UNSAT
  end if
else
  $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
  if $t =$ UNSAT **then**
    $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
  else
    $\rho :=$ true
  end if
end if
return $\langle t, \rho \rangle$

ExecuteSymbolic:

- $\phi_1 = x_0 > 0$
- $\mathcal{S} = \{x \mapsto x_0, y \mapsto 4\}$
- $\phi_2 = true$

Input and exit constraints:

- $\phi = \phi_{ic}$
- $\phi' = \phi_{ec}$

ExtendFrontier – The Idea

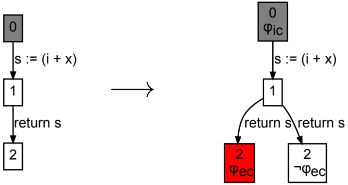
$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
    let $\langle \Sigma, \sigma^I, \rightarrow \rangle =$ GetProc(Edge(S_{k-1}, S_k))
    $\phi :=$ InputContraints$(\mathcal{S})$
    $\phi' := S_k[e/x]$
    $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')$
    if $r =$ FAIL **then**
        $t := m$
        $\rho :=$ true
    else
        $\rho :=$ ComputeRefinePred(m)
        $t :=$ UNSAT
    end if
else
    $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
    if $t =$ UNSAT **then**
        $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
    else
        $\rho :=$ true
    end if
end if
return $\langle t, \rho \rangle$

ExecuteSymbolic:

- $\phi_1 = x_0 > 0$
- $\mathcal{S} = \{x \mapsto x_0, y \mapsto 4\}$
- $\phi_2 = true$

Input and exit constraints:

- $\phi = \phi_{ic}$
- $\phi' = \phi_{ec}$



ExtendFrontier – The Idea

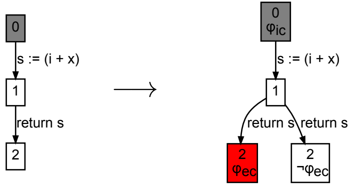
$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
  let $\langle \Sigma, \sigma^I, \to \rangle =$ GetProc(Edge(S_{k-1}, S_k))
  $\phi :=$ InputContraints$(\mathcal{S})$
  $\phi' := S_k[e/x]$
  $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \to \rangle, \neg\phi')$
  if $r = \text{FAIL}$ **then**
    $t := m$
    $\rho :=$ true
  else
    $\rho :=$ ComputeRefinePred(m)
    $t := \text{UNSAT}$
  end if
else
  $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
  if $t = \text{UNSAT}$ **then**
    $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
  else
    $\rho :=$ true
  end if
end if
return $\langle t, \rho \rangle$

ExecuteSymbolic:

- $\phi_1 = x_0 > 0$
- $\mathcal{S} = \{x \mapsto x_0, y \mapsto 4\}$
- $\phi_2 = true$

Input and exit constraints:

- $\phi = \phi_{ic}$
- $\phi' = \phi_{ec}$



ExtendFrontier – The Idea

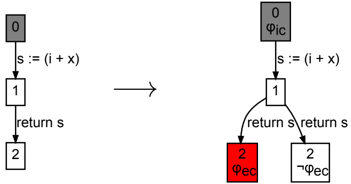
$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
  let $\langle \Sigma, \sigma^I, \to \rangle =$ GetProc(Edge(S_{k-1}, S_k))
  $\phi :=$ InputContraints$(\mathcal{S})$
  $\phi' := S_k[e/x]$
  $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \to \rangle, \neg\phi')$
  if $r =$ FAIL **then**
    $t := m$
    $\rho :=$ true
  else
    $\rho :=$ ComputeRefinePred(m)
    $t :=$ UNSAT
  end if
else
  $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
  if $t =$ UNSAT **then**
    $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
  else
    $\rho :=$ true
  end if
end if
return $\langle t, \rho \rangle$

ExecuteSymbolic:

- $\phi_1 = x_0 > 0$
- $\mathcal{S} = \{x \mapsto x_0, y \mapsto 4\}$
- $\phi_2 = true$

Input and exit constraints:

- $\phi = \phi_{ic}$
- $\phi' = \phi_{ec}$



InputConstraints

$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
    let $\langle \Sigma, \sigma^I, \rightarrow \rangle =$ GetProc(Edge(S_{k-1}, S_k))
    $\phi :=$ InputContraints$(\mathcal{S})$
    $\phi' := S_k[e/x]$
    $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg\phi')$
    if $r =$ FAIL **then**
        $t := m$
        $\rho :=$ true
    else
        $\rho :=$ ComputeRefinePred(m)
        $t :=$ UNSAT
    end if
else
    $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
    if $t =$ UNSAT **then**
        $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
    else
        $\rho :=$ true
    end if
end if
return $\langle t, \rho \rangle$

- InputConstraints takes the symbolic map $\mathcal{S}$ as the only parameter. They write:
  - *The predicate ϕ corresponds to the constraints on Q's input variables which are computed directly from the symbolic map $\mathcal{S}$ (by the auxiliary function InputConstraints [...])*
- When FAIL is reported, the test input m is returned unmodified.

InputConstraints implementation

Given the below quote, how do we implement InputConstraints?

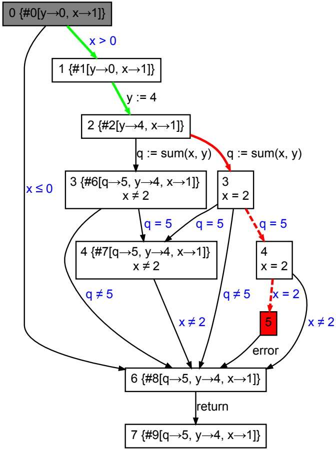
- *The predicate ϕ corresponds to the constraints on Q's input variables which are computed directly from the symbolic map $\mathcal{S}$ (by the auxiliary function InputConstraints [...])*

$$v := f(a_1, \ldots, a_n)$$

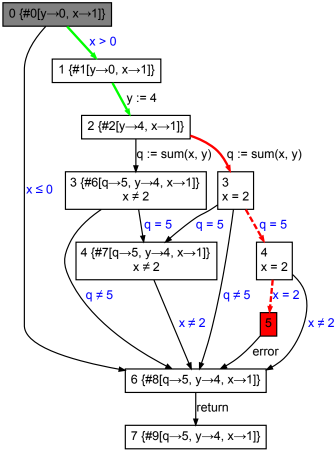
`int f(int p1, ..., int pn) { ... }`

$$\phi_{ic} := \text{InputConstraints}(\mathcal{S})$$
$$:= \left(\bigwedge_{p_i \in \text{params}(Q)} p_i = \text{SymbolicEval}(a_i, \mathcal{S}) \right)$$
$$:= p_1 = \text{SymbolicEval}(a_1, \mathcal{S}) \wedge$$
$$\ldots \wedge$$
$$p_n = \text{SymbolicEval}(a_n, \mathcal{S})$$

Example

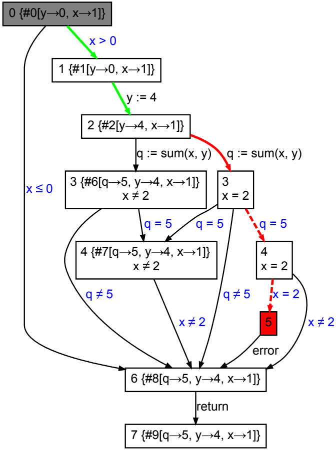

0 {#0[y→0, x→1]}
x > 0
1 {#1[y→0, x→1]}
y := 4
2 {#2[y→4, x→1]}
q := sum(x, y)
q := sum(x, y)
3 {#6[q→5, y→4, x→1]} x ≠ 2
3 x = 2
x ≤ 0
q = 5
q = 5
q = 5
4 {#7[q→5, y→4, x→1]} x ≠ 2
4 x = 2
q ≠ 5
q ≠ 5
x = 2
x ≠ 2
x ≠ 2
5
error
6 {#8[q→5, y→4, x→1]}
return
7 {#9[q→5, y→4, x→1]}

Example



$$\mathcal{S} = \{x \mapsto x_0, y \mapsto 4\}$$

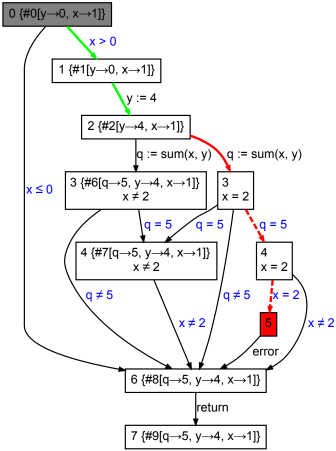
Example



$\mathcal{S} = \{x \mapsto x_0, y \mapsto 4\}$

```
int sum(int i, int x)
{
  int s = i + x;
  return s;
}
```

Example



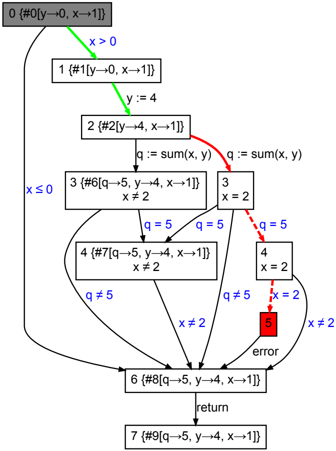
$\mathcal{S} = \{x \mapsto x_0, y \mapsto 4\}$

```
int sum(int i, int x)
{
  int s = i + x;
  return s;
}
```

$$\phi_{ic} := \text{InputConstraints}(\mathcal{S})$$

$$:= \left(\bigwedge_{p_i \in \text{params}(Q)} p_i = \text{SymbolicEval}(a_i, \mathcal{S}) \right)$$

Example



$$\mathcal{S} = \{x \mapsto x_0, y \mapsto 4\}$$

```
int sum(int i, int x)
{
  int s = i + x;
  return s;
}
```

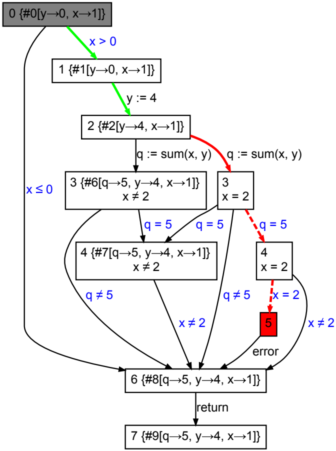
$$\phi_{ic} := \text{InputConstraints}(\mathcal{S})$$

$$:= \left(\bigwedge_{p_i \in \text{params}(Q)} p_i = \text{SymbolicEval}(a_i, \mathcal{S}) \right)$$

$$:= i = \text{SymbolicEval}(x, \mathcal{S}) \wedge$$

$$x = \text{SymbolicEval}(y, \mathcal{S})$$

Example



$\mathcal{S} = \{x \mapsto x_0, y \mapsto 4\}$

```
int sum(int i, int x)
{
  int s = i + x;
  return s;
}
```

$$\phi_{ic} := \text{InputConstraints}(\mathcal{S})$$

$$:= \left(\bigwedge_{p_i \in \text{params}(Q)} p_i = \text{SymbolicEval}(a_i, \mathcal{S}) \right)$$

$$:= i = \text{SymbolicEval}(x, \mathcal{S}) \wedge$$

$$x = \text{SymbolicEval}(y, \mathcal{S})$$

$$:= i = x_0 \wedge x = 4$$

Missing path constraint examples – TestAbs

```
int abs(int a)
{
  if (a < 0)
    return -a;
  return a;
}
```

- Returns -2147483648 when given as input.

Missing path constraint examples – TestAbs

```
int abs(int a)
{
  if (a < 0)
    return -a;
  return a;
}
```

- Returns -2147483648 when given as input.

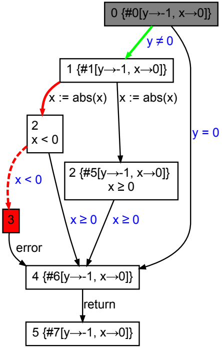
```
void testabs(int x, int y)
{
  if(y ≠ 0)
  {
    x = abs(x);
    if(x < 0)
      error;
  }
}
```

Missing path constraint examples – TestAbs

```
int abs(int a)
{
  if (a < 0)
    return -a;
  return a;
}
```

```
void testabs(int x, int y)
{
  if(y ≠ 0)
  {
    x = abs(x);
    if(x < 0)
      error;
  }
}
```

- Returns -2147483648 when given as input.



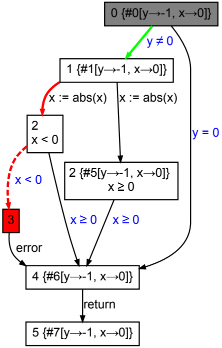
Missing path constraint examples – TestAbs

```
int abs(int a)
{
  if (a < 0)
    return -a;
  return a;
}
```

- Returns −2147483648 when given as input.

```
void testabs(int x, int y)
{
  if(y ≠ 0)
  {
    x = abs(x);
    if(x < 0)
      error;
  }
}
```

- Solution returned by abs does not mention y.



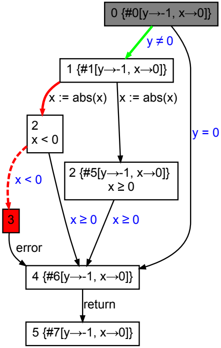
Missing path constraint examples – TestAbs

```
int abs(int a)
{
  if (a < 0)
    return -a;
  return a;
}
```

- Returns −2147483648 when given as input.

```
void testabs(int x, int y)
{
  if(y ≠ 0)
  {
    x = abs(x);
    if(x < 0)
      error;
  }
}
```

- Solution returned by abs does not mention *y*.
- Call SAT solver again to find value for *y*!



Missing path constraint examples – TestAbs2

```
int abs(int a)
{
  if (a < 0)
    return -a;
  return a;
}
```

- Returns -2147483648 when given as input.

Missing path constraint examples – TestAbs2

```
int abs(int a)
{
  if (a < 0)
    return -a;
  return a;
}
```

- Returns -2147483648 when given as input.

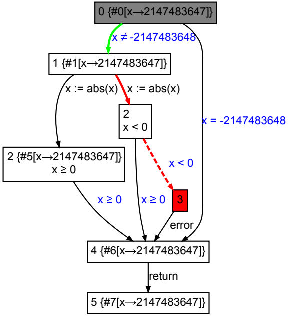
```
void testabs2(int x)
{
  if(x ≠ -2147483648)
  {
    x = abs(x);
    if(x < 0)
      error;
  }
}
```

Missing path constraint examples – TestAbs2

```
int abs(int a)
{
  if (a < 0)
    return -a;
  return a;
}
```

```
void testabs2(int x)
{
  if(x ≠ -2147483648)
  {
    x = abs(x);
    if(x < 0)
      error;
  }
}
```

- Returns -2147483648 when given as input.



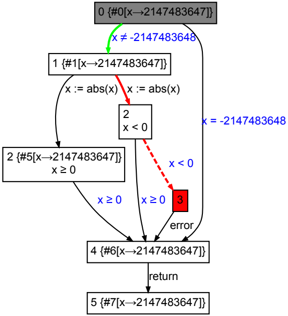
Missing path constraint examples – TestAbs2

```
int abs(int a)
{
  if (a < 0)
    return -a;
  return a;
}
```

- Returns -2147483648 when given as input.

```
void testabs2(int x)
{
  if(x ≠ -2147483648)
  {
    x = abs(x);
    if(x < 0)
      error;
  }
}
```

- $x = -2147483648$, which is prohibited!



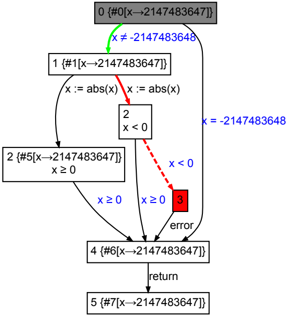
Missing path constraint examples – TestAbs2

```
int abs(int a)
{
  if (a < 0)
    return -a;
  return a;
}
```

- Returns −2147483648 when given as input.

```
void testabs2(int x)
{
  if(x ≠ -2147483648)
  {
    x = abs(x);
    if(x < 0)
      error;
  }
}
```

- $x = -2147483648$, which is prohibited!
- Must include path constraint when analyzing abs.



Adding the path constraint ϕ_1

$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
  let $\langle \Sigma, \sigma^I, \rightarrow \rangle =$ GetProc(Edge(S_{k-1}, S_k))
  $\phi :=$ InputContraints$(\mathcal{S}, \phi_1)$
  $\phi' := S_k[e/x]$
  $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')$
  if $r =$ FAIL **then**
    $t := m$
    $\rho :=$ true
  else
    $\rho :=$ ComputeRefinePred(m)
    $t :=$ UNSAT
  end if
else
  $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
  if $t =$ UNSAT **then**
    $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
  else
    $\rho :=$ true
  end if
end if
return $\langle t, \rho \rangle$

Include the path constraint ϕ_1:

$$\phi_{ic} := \text{InputConstraints}(\mathcal{S}, \phi_1)$$

$$:= \left(\bigwedge_{p_i \in \text{params}(Q)} p_i = \text{SymbolicEval}(a_i, \mathcal{S}) \right)$$

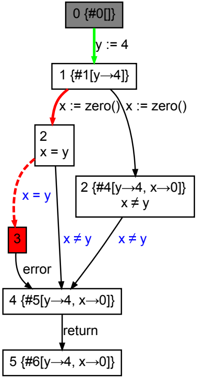
$$\wedge\ \phi_1$$

Missing symbolic variables in solution

```
void foo()
{
  int y = 4;
  int x = zero();
  if(x == y)
    error;
}

int zero()
{
  return 0;
}
```

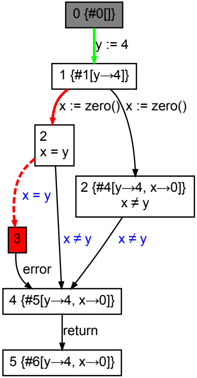
Missing symbolic variables in solution

```
void foo()
{
  int y = 4;
  int x = zero()
  if(x == y)
    error;
}

int zero()
{
  return 0;
}
```



Missing symbolic variables in solution

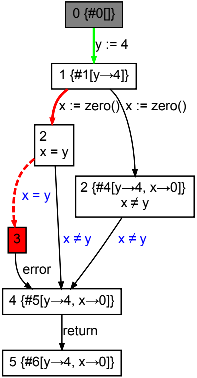
```
void foo()
{
  int y = 4;
  int x = zero()
  if(x == y)
    error;
}

int zero()
{
  return 0;
}
```



- Input constraint does not include y since it is not part of the arguments to zero.

Missing symbolic variables in solution

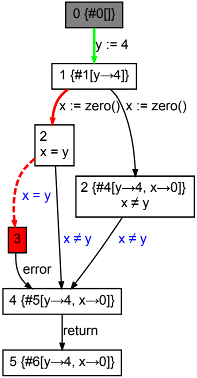
```
void foo()
{
  int y = 4;
  int x = zero()
  if(x == y)
    error;
}

int zero()
{
  return 0;
}
```



- Input constraint does not include y since it is not part of the arguments to zero.
- The analysis of zero does not know that y must be 4 and returns a solution where $y \mapsto 0$!

Missing symbolic variables in solution

```
void foo()
{
  int y = 4;
  int x = zero()
  if(x == y)
    error;
}

int zero()
{
  return 0;
}
```



- Input constraint does not include y since it is not part of the arguments to zero.
- The analysis of zero does not know that y must be 4 and returns a solution where $y \mapsto 0$!
- Must include all symbolic variables in constraint, such that zero knows how they are bound.

Other problems with InputConstraints

- Variables used in the input constraint must be linked together with the exit constraint.

Other problems with InputConstraints

- Variables used in the input constraint must be linked together with the exit constraint. $\mapsto$ Need a two-step process when constructing the input constraint.

Other problems with InputConstraints

- Variables used in the input constraint must be linked together with the exit constraint. $\mapsto$ Need a two-step process when constructing the input constraint.
- What about renaming? When x is used both in the caller and called procedure?

Other problems with InputConstraints

- Variables used in the input constraint must be linked together with the exit constraint. $\mapsto$ Need a two-step process when constructing the input constraint.
- What about renaming? When x is used both in the caller and called procedure? $\mapsto$ Rename external variables using a renamer π: $\pi(a) = 1{\downarrow}a$

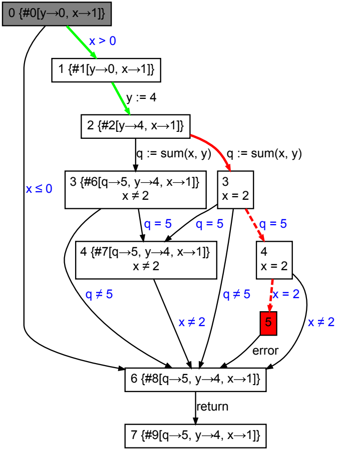
The fix to InputConstraints

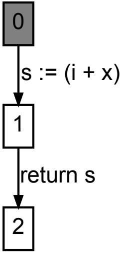
$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle :=$ GetWholeAbstractTrace(τ_o, F)
$(k-1, k) :=$ Frontier(τ_w)
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle :=$ ExecuteSymbolic(τ_w, P)
if Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
    let $\langle \Sigma, \sigma^I, \rightarrow \rangle =$ GetProc(Edge(S_{k-1}, S_k))
    $\phi :=$ InputContraints$(\mathcal{S}, \phi_1)$
    $\phi' := S_k[e/x]$
    $\langle r, m \rangle :=$ DASH$(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')$
    if $r =$ FAIL **then**
        $t := m$
        $\rho :=$ true
    else
        $\rho :=$ ComputeRefinePred(m)
        $t :=$ UNSAT
    end if
else
    $t :=$ IsSAT$(\phi_1, \mathcal{S}, \phi_2, P)$
    if $t =$ UNSAT **then**
        $\rho :=$ RefinePred$(\mathcal{S}, \tau_w)$
    else
        $\rho :=$ true
    end if
end if
return $\langle t, \rho \rangle$

$$\phi_{ic} := \text{InputConstraints}(\mathcal{S}, \phi_1)$$

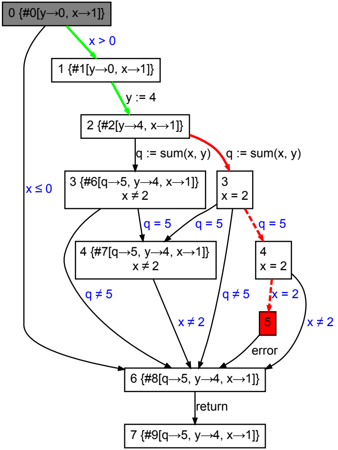
$$:= \left(\bigwedge_{p_i \in \text{params}(Q)} p_i = \pi(a_i) \right) \wedge \left(\bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e) \right) \wedge \pi(\phi_1)$$

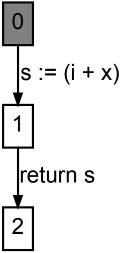
Input- and exit-constraints on analyzed graph



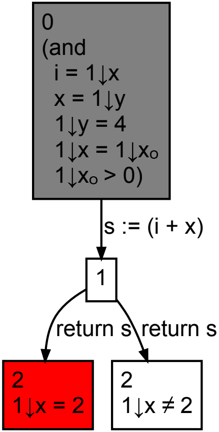


Input- and exit-constraints on analyzed graph





⟶



Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

```
τ_w = ⟨S_0, S_1, ..., S_n⟩ := GetWholeAbstractTrace(τ_o, F)
(k − 1, k) := Frontier(τ_w)
⟨φ_1, S, φ_2⟩ := ExecuteSymbolic(τ_w, P)
if Edge(S_{k−1}, S_k) ∈ CallReturn(E) then
    let ⟨Σ, σ^I, →⟩ = GetProc(Edge(S_{k−1}, S_k))
    φ := InputContraints(S, φ_1)
    φ' := S_k[e/x]
    ⟨r, m⟩ := DASH(⟨Σ, σ^I ∧ φ, →⟩, ¬φ')
    if r = FAIL then
        t := m
        ρ := true
    else
        ρ := ComputeRefinePred(m)
        t := UNSAT
    end if
else
    t := IsSAT(φ_1, S, φ_2, P)
    if t = UNSAT then
        ρ := RefinePred(S, τ_w)
    else
        ρ := true
    end if
end if
return ⟨t, ρ⟩
```

Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- Initial symbolic variables v_0 for all parameters of P'.

Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- Initial symbolic variables v_0 for all parameters of P'.
- Values for the variables mentioned in the input constraint ϕ_{ic}:

Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- Initial symbolic variables v_0 for all parameters of P'.
- Values for the variables mentioned in the input constraint ϕ_{ic}:
  - Parameters and local variables for P.

Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- Initial symbolic variables v_0 for all parameters of P'.
- Values for the variables mentioned in the input constraint ϕ_{ic}:
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.

Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- Initial symbolic variables v_0 for all parameters of P'.
- Values for the variables mentioned in the input constraint ϕ_{ic}:
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - Variables mentioned in the input constraint for P (if any).

Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- Initial symbolic variables v_0 for all parameters of P'.
- Values for the variables mentioned in the input constraint ϕ_{ic}:
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - Variables mentioned in the input constraint for P (if any).

Only some parts are needed.

Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- Initial symbolic variables v_0 for all parameters of P'.
- Values for the variables mentioned in the input constraint ϕ_{ic}:
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - Variables mentioned in the input constraint for P (if any).

Only some parts are needed.

Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- Initial symbolic variables v_0 for all parameters of P'.
- Values for the variables mentioned in the input constraint ϕ_{ic}:
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - Variables mentioned in the input constraint for P (if any).

Only some parts are needed.

Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- Initial symbolic variables v_0 for all parameters of P'.
- Values for the variables mentioned in the input constraint ϕ_{ic}:
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - Variables mentioned in the input constraint for P (if any).

Only some parts are needed.

$t := \quad m$

Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- Initial symbolic variables v_0 for all parameters of P'.
- Values for the variables mentioned in the input constraint ϕ_{ic}:
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - Variables mentioned in the input constraint for P (if any).

Only some parts are needed.

$$t := \qquad m \setminus \{v_0 \mid \forall v \in \text{params}(P')\}$$

Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- Initial symbolic variables v_0 for all parameters of P'.
- Values for the variables mentioned in the input constraint ϕ_{ic}:
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - Variables mentioned in the input constraint for P (if any).

Only some parts are needed.

$$t := \pi^{-1}\Big(m \setminus \{v_0 \mid \forall v \in \mathrm{params}(P')\}\Big)$$

Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- Initial symbolic variables v_0 for all parameters of P'.
- Values for the variables mentioned in the input constraint ϕ_{ic}:
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - Variables mentioned in the input constraint for P (if any).

Only some parts are needed.

$$t := \pi^{-1}\Big(m \setminus \{v_0 \mid \forall v \in \text{params}(P')\}\Big) \setminus \Big(\text{locals}(P) \cup \text{params}(P)\Big)$$

ComputeRefinePred

Goal: Extract refinement predicate ρ from proof m that shows that S_k cannot be reached in P'.

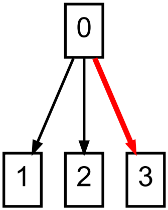
$\tau_w = \langle S_0, S_1, \ldots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_o, F)$
$(k-1, k) := \text{Frontier}(\tau_w)$
$\langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)$
if $\text{Edge}(S_{k-1}, S_k) \in \text{CallReturn}(E)$ **then**
    let $\langle \Sigma, \sigma^I, \to \rangle = \text{GetProc}(\text{Edge}(S_{k-1}, S_k))$
    $\phi := \text{InputContraints}(\mathcal{S}, \phi_1)$
    $\phi' := S_k[e/x]$
    $\langle r, m \rangle := \text{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \to \rangle, \neg\phi')$
    if $r = \textsc{fail}$ **then**
        $t := m$
        $\rho := \text{true}$
    else
        $\rho := \text{ComputeRefinePred}(m)$
        $t := \textsc{unsat}$
    end if
else
    $t := \text{IsSAT}(\phi_1, \mathcal{S}, \phi_2, P)$
    if $t = \textsc{unsat}$ **then**
        $\rho := \text{RefinePred}(\mathcal{S}, \tau_w)$
    else
        $\rho := \text{true}$
    end if
end if
return $\langle t, \rho \rangle$

ComputeRefinePred

Goal: Extract refinement predicate ρ from proof m that shows that S_k cannot be reached in P'.

- Predicate $\neg p_i$ removes edge to some error region in P'

Refinement of initial region in P':

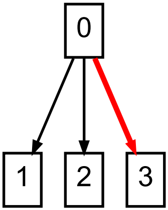


ComputeRefinePred

Goal: Extract refinement predicate ρ from proof m that shows that S_k cannot be reached in P'.

- Predicate $\neg p_i$ removes edge to some error region in $P' \mapsto$ a path to region S_k in P.

Refinement of initial region in P':



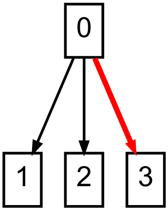
ComputeRefinePred

Goal: Extract refinement predicate ρ from proof m that shows that S_k cannot be reached in P'.

- Predicate $\neg p_i$ removes edge to some error region in $P' \mapsto$ a path to region S_k in P.
- The conjunction of these $\neg p_1 \wedge \ldots \wedge \neg p_n$ removes all edges to error regions in P'

.

Refinement of initial region in P':

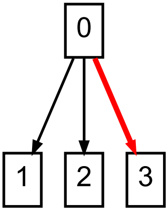


ComputeRefinePred

Goal: Extract refinement predicate ρ from proof m that shows that S_k cannot be reached in P'.

- Predicate $\neg p_i$ removes edge to some error region in $P' \mapsto$ a path to region S_k in P.
- The conjunction of these $\neg p_1 \wedge \ldots \wedge \neg p_n$ removes all edges to error regions in $P' \mapsto$ every path to region S_k in P.

Refinement of initial region in P':

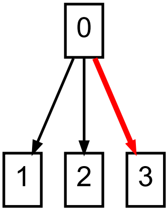


ComputeRefinePred

Goal: Extract refinement predicate ρ from proof m that shows that S_k cannot be reached in P'.

- Predicate $\neg p_i$ removes edge to some error region in $P' \mapsto$ a path to region S_k in P.
- The conjunction of these $\neg p_1 \wedge \ldots \wedge \neg p_n$ removes all edges to error regions in $P' \mapsto$ every path to region S_k in P.

Refinement of initial region in P':



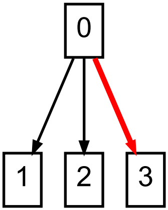
We need the negated predicate:
$\neg(\neg p_1 \wedge \ldots \wedge \neg p_n) = p_1 \vee \ldots \vee p_n$

ComputeRefinePred

Goal: Extract refinement predicate ρ from proof m that shows that S_k cannot be reached in P'.

- Predicate $\neg p_i$ removes edge to some error region in $P' \mapsto$ a path to region S_k in P.
- The conjunction of these $\neg p_1 \wedge \ldots \wedge \neg p_n$ removes all edges to error regions in $P' \mapsto$ every path to region S_k in P.

Refinement of initial region in P':



We need the negated predicate:
$\neg(\neg p_1 \wedge \ldots \wedge \neg p_n) = p_1 \vee \ldots \vee p_n$

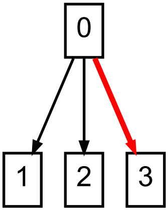
$$\rho := \bigvee_{\rho_i \in \text{InitialRefines}(m)} \rho_i$$

ComputeRefinePred

Goal: Extract refinement predicate ρ from proof m that shows that S_k cannot be reached in P'.

- Predicate $\neg p_i$ removes edge to some error region in $P' \mapsto$ a path to region S_k in P.
- The conjunction of these $\neg p_1 \wedge \ldots \wedge \neg p_n$ removes all edges to error regions in $P' \mapsto$ every path to region S_k in P.

Refinement of initial region in P':



We need the negated predicate:
$\neg(\neg p_1 \wedge \ldots \wedge \neg p_n) = p_1 \vee \ldots \vee p_n$

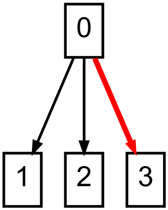
$$\rho := \left(\bigvee_{\rho_i \in \text{InitialRefines}(m)} \rho_i \right) \left[a_0/v_0, \ldots, a_n/v_n \right] \mid v_i \in \text{params}(P')$$

ComputeRefinePred

Goal: Extract refinement predicate ρ from proof m that shows that S_k cannot be reached in P'.

- Predicate $\neg p_i$ removes edge to some error region in $P' \mapsto$ a path to region S_k in P.
- The conjunction of these $\neg p_1 \wedge \ldots \wedge \neg p_n$ removes all edges to error regions in $P' \mapsto$ every path to region S_k in P.

Refinement of initial region in P':



We need the negated predicate:
$\neg(\neg p_1 \wedge \ldots \wedge \neg p_n) = p_1 \vee \ldots \vee p_n$

$$\rho := \pi^{-1}\left(\left(\bigvee_{\rho_i \in \text{InitialRefines}(m)} \rho_i\right)\left[a_0/v_0, \ldots, a_n/v_n\right] \;\middle|\; v_i \in \text{params}(P')\right)$$

ExtendFrontier implementation

- Renaming
- InputConstraints
- ExitConstraints
- Graph construction
- Extract test input
- Extract refinement predicate

let $\langle S_{k-1}, \_\rangle = RS_{k-1}$
$\langle \phi, \mathcal{S}\rangle := \text{ExecuteSymbolic}(\tau_c, P, \mathcal{P}, \mathcal{G})$
$op := \text{Op}(S_{k-1}, S_k)$
if op **matches** $\mathsf{v} := \mathsf{f}(a_0, \ldots, a_n)$ **then**
  let $\langle \rho_k, \_\rangle = S_k$
  $P' := \text{Lookup}(\mathsf{f}, \mathcal{P})$
  $\pi := \text{CreateVariableRenamer}(\text{locals}(P) \cup \{v, v_0 \mid \forall v \in \text{params}(P)\})$
  $\phi_{ic} := \left(\bigwedge_{v_i \in \text{params}(P')} v_i = \pi(a_i)\right) \wedge \left(\bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e)\right) \wedge \pi(\phi)$
  $\phi_{ec} := \pi(\rho_k[@r/\mathsf{v}])$
  $\mathcal{G}' := \text{ReconstructGraphsAndInsertConstraints}(\mathcal{G}, \phi_{ic}, \phi_{ec}, P')$
  $\langle r, z\rangle = \text{DashLoop}(\mathcal{G}', \mathcal{P}, P')$
  if $r = \text{FAIL}$ **then**
    $t := \pi^{-1}\left(z \setminus \{v_0 \mid \forall v \in \text{params}(P')\}\right) \setminus \left(\text{locals}(P) \cup \text{params}(P)\right)$
    $\rho := \text{true}$
  else
    $t := \text{UNSAT}$
    $\rho := \pi^{-1}\left(\left(\bigvee_{\rho_i \in \text{InitialRefines}(z)} \rho_i\right)[a_0/v_0, \ldots, a_n/v_n] \mid v_i \in \text{params}(P')\right)$
  end if
else
  $t := \text{IsSAT}(\phi, P)$
  if $t = \text{UNSAT}$ **then**
    $\rho := \text{RefinePred}(\tau_c)$
  else
    $\rho := \text{true}$
  end if
end if
return $\langle t, \rho\rangle$

ExtendFrontier implementation

- Renaming
- InputConstraints
- ExitConstraints
- Graph construction
- Extract test input
- Extract refinement predicate

let $\langle S_{k-1}, \_ \rangle = RS_{k-1}$
$\langle \phi, \mathcal{S} \rangle :=$ ExecuteSymbolic$(\tau_c, P, \mathcal{P}, \mathcal{G})$
$op :=$ Op(S_{k-1}, S_k)
if op **matches** $\mathsf{v} := \mathsf{f}(a_0, \ldots, a_n)$ **then**
  let $\langle \rho_k, \_ \rangle = S_k$
  $P' :=$ Lookup$(\mathsf{f}, \mathcal{P})$
  $\pi :=$ CreateVariableRenamer(locals$(P) \cup \{v, v_0 \mid \forall v \in$ params$(P)\})$
  $\phi_{ic} := \left(\bigwedge_{v_i \in \text{params}(P')} v_i = \pi(a_i) \right) \wedge \left(\bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e) \right) \wedge \pi(\phi)$
  $\phi_{ec} := \pi(\rho_k[@r/\mathsf{v}])$
  $\mathcal{G}' :=$ ReconstructGraphsAndInsertConstraints$(\mathcal{G}, \phi_{ic}, \phi_{ec}, P')$
  $\langle r, z \rangle =$ DashLoop$(\mathcal{G}', \mathcal{P}, P')$
  if $r = \text{FAIL}$ **then**
    $t := \pi^{-1}\left(z \setminus \{v_0 \mid \forall v \in \text{params}(P')\}\right) \setminus \left(\text{locals}(P) \cup \text{params}(P)\right)$
    $\rho :=$ true
  else
    $t := \text{UNSAT}$
    $\rho := \pi^{-1}\left(\left(\bigvee_{\rho_i \in \text{InitialRefines}(z)} \rho_i\right) [a_0/v_0, \ldots, a_n/v_n] \mid v_i \in \text{params}(P')\right)$
  end if
else
  $t :=$ IsSAT(ϕ, P)
  if $t = \text{UNSAT}$ **then**
    $\rho :=$ RefinePred(τ_c)
  else
    $\rho :=$ true
  end if
end if
return $\langle t, \rho \rangle$

ExtendFrontier implementation

- Renaming
- InputConstraints
- ExitConstraints
- Graph construction
- Extract test input
- Extract refinement predicate

let $\langle S_{k-1}, \_\rangle = RS_{k-1}$
$\langle \phi, \mathcal{S}\rangle := \text{ExecuteSymbolic}(\tau_c, P, \mathcal{P}, \mathcal{G})$
$op := \text{Op}(S_{k-1}, S_k)$
if op **matches** $\mathsf{v} := \mathsf{f}(a_0, \ldots, a_n)$ **then**
  let $\langle \rho_k, \_\rangle = S_k$
  $P' := \text{Lookup}(\mathsf{f}, \mathcal{P})$
  $\pi := \text{CreateVariableRenamer}(\text{locals}(P) \cup \{v, v_0 \mid \forall v \in \text{params}(P)\})$
  $\phi_{ic} := \left(\bigwedge_{v_i \in \text{params}(P')} v_i = \pi(a_i)\right) \wedge \left(\bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e)\right) \wedge \pi(\phi)$
  $\phi_{ec} := \pi(\rho_k[@r/\mathsf{v}])$
  $\mathcal{G}' := \text{ReconstructGraphsAndInsertConstraints}(\mathcal{G}, \phi_{ic}, \phi_{ec}, P')$
  $\langle r, z\rangle = \text{DashLoop}(\mathcal{G}', \mathcal{P}, P')$
  if $r = \text{FAIL}$ **then**
    $t := \pi^{-1}\left(z \setminus \{v_0 \mid \forall v \in \text{params}(P')\}\right) \setminus \left(\text{locals}(P) \cup \text{params}(P)\right)$
    $\rho := \text{true}$
  else
    $t := \text{UNSAT}$
    $\rho := \pi^{-1}\left(\left(\bigvee_{\rho_i \in \text{InitialRefines}(z)} \rho_i\right) [a_0/v_0, \ldots, a_n/v_n] \;\mid v_i \in \text{params}(P')\right)$
  end if
else
  $t := \text{IsSAT}(\phi, P)$
  if $t = \text{UNSAT}$ **then**
    $\rho := \text{RefinePred}(\tau_c)$
  else
    $\rho := \text{true}$
  end if
end if
return $\langle t, \rho\rangle$

ExtendFrontier implementation

- Renaming
- InputConstraints
- ExitConstraints
- Graph construction
- Extract test input
- Extract refinement predicate

let $\langle S_{k-1}, \_\rangle = RS_{k-1}$
$\langle \phi, \mathcal{S}\rangle := \text{ExecuteSymbolic}(\tau_c, P, \mathcal{P}, \mathcal{G})$
$op := \text{Op}(S_{k-1}, S_k)$
if op **matches** $\mathsf{v} := \mathsf{f}(a_0, \ldots, a_n)$ **then**
    let $\langle \rho_k, \_\rangle = S_k$
    $P' := \text{Lookup}(\mathsf{f}, \mathcal{P})$
    $\pi := \text{CreateVariableRenamer}(\text{locals}(P) \cup \{v, v_0 \mid \forall v \in \text{params}(P)\})$
    $\phi_{ic} := \left(\bigwedge_{v_i \in \text{params}(P')} v_i = \pi(a_i)\right) \wedge \left(\bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e)\right) \wedge \pi(\phi)$
    $\phi_{ec} := \pi(\rho_k[@r/\mathsf{v}])$
    $\mathcal{G}' := \text{ReconstructGraphsAndInsertConstraints}(\mathcal{G}, \phi_{ic}, \phi_{ec}, P')$
    $\langle r, z\rangle = \text{DashLoop}(\mathcal{G}', \mathcal{P}, P')$
    if $r = \text{FAIL}$ **then**
        $t := \pi^{-1}\left(z \setminus \{v_0 \mid \forall v \in \text{params}(P')\}\right) \setminus \left(\text{locals}(P) \cup \text{params}(P)\right)$
        $\rho := \text{true}$
    else
        $t := \text{UNSAT}$
        $\rho := \pi^{-1}\left(\left(\bigvee_{\rho_i \in \text{InitialRefines}(z)} \rho_i\right)[a_0/v_0, \ldots, a_n/v_n] \mid v_i \in \text{params}(P')\right)$
    end if
else
    $t := \text{IsSAT}(\phi, P)$
    if $t = \text{UNSAT}$ **then**
        $\rho := \text{RefinePred}(\tau_c)$
    else
        $\rho := \text{true}$
    end if
end if
return $\langle t, \rho\rangle$

ExtendFrontier implementation

- Renaming
- InputConstraints
- ExitConstraints
- Graph construction
- Extract test input
- Extract refinement predicate

let $\langle S_{k-1}, \_ \rangle = RS_{k-1}$
$\langle \phi, \mathcal{S} \rangle := \text{ExecuteSymbolic}(\tau_c, P, \mathcal{P}, \mathcal{G})$
$op := \text{Op}(S_{k-1}, S_k)$
if op **matches** $\mathsf{v} := \mathsf{f}(a_0, \ldots, a_n)$ **then**
    let $\langle \rho_k, \_ \rangle = S_k$
    $P' := \text{Lookup}(\mathsf{f}, \mathcal{P})$
    $\pi := \text{CreateVariableRenamer}(\text{locals}(P) \cup \{v, v_0 \mid \forall v \in \text{params}(P)\})$
    $\phi_{ic} := \left(\bigwedge_{v_i \in \text{params}(P')} v_i = \pi(a_i) \right) \wedge \left(\bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e) \right) \wedge \pi(\phi)$
    $\phi_{ec} := \pi(\rho_k[@r/\mathsf{v}])$
    $\mathcal{G}' := \text{ReconstructGraphsAndInsertConstraints}(\mathcal{G}, \phi_{ic}, \phi_{ec}, P')$
    $\langle r, z \rangle = \text{DashLoop}(\mathcal{G}', \mathcal{P}, P')$
    if $r = \text{FAIL}$ **then**
        $t := \pi^{-1}\left(z \setminus \{v_0 \mid \forall v \in \text{params}(P')\}\right) \setminus \left(\text{locals}(P) \cup \text{params}(P)\right)$
        $\rho := \text{true}$
    else
        $t := \text{UNSAT}$
        $\rho := \pi^{-1}\left(\left(\bigvee_{\rho_i \in \text{InitialRefines}(z)} \rho_i\right) [a_0/v_0, \ldots, a_n/v_n] \mid v_i \in \text{params}(P')\right)$
    end if
else
    $t := \text{IsSAT}(\phi, P)$
    if $t = \text{UNSAT}$ **then**
        $\rho := \text{RefinePred}(\tau_c)$
    else
        $\rho := \text{true}$
    end if
end if
return $\langle t, \rho \rangle$

ExtendFrontier implementation

- Renaming
- InputConstraints
- ExitConstraints
- Graph construction
- Extract test input
- Extract refinement predicate

let $\langle S_{k-1}, \_\rangle = RS_{k-1}$
$\langle \phi, \mathcal{S}\rangle := \text{ExecuteSymbolic}(\tau_c, P, \mathcal{P}, \mathcal{G})$
$op := \text{Op}(S_{k-1}, S_k)$
if op **matches** $\mathtt{v} := \mathtt{f}(a_0, \ldots, a_n)$ **then**
  let $\langle \rho_k, \_\rangle = S_k$
  $P' := \text{Lookup}(\mathtt{f}, \mathcal{P})$
  $\pi := \text{CreateVariableRenamer}(\text{locals}(P) \cup \{v, v_0 \mid \forall v \in \text{params}(P)\})$
  $\phi_{ic} := \left(\bigwedge_{v_i \in \text{params}(P')} v_i = \pi(a_i)\right) \wedge \left(\bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e)\right) \wedge \pi(\phi)$
  $\phi_{ec} := \pi(\rho_k[@r/\mathtt{v}])$
  $\mathcal{G}' := \text{ReconstructGraphsAndInsertConstraints}(\mathcal{G}, \phi_{ic}, \phi_{ec}, P')$
  $\langle r, z\rangle = \text{DashLoop}(\mathcal{G}', \mathcal{P}, P')$
  if $r = \text{FAIL}$ **then**
    $t := \pi^{-1}\left(z \setminus \{v_0 \mid \forall v \in \text{params}(P')\}\right) \setminus \left(\text{locals}(P) \cup \text{params}(P)\right)$
    $\rho := \text{true}$
  else
    $t := \text{UNSAT}$
    $\rho := \pi^{-1}\left(\left(\bigvee_{\rho_i \in \text{InitialRefines}(z)} \rho_i\right) [a_0/v_0, \ldots, a_n/v_n] \mid v_i \in \text{params}(P')\right)$
  end if
else
  $t := \text{IsSAT}(\phi, P)$
  if $t = \text{UNSAT}$ **then**
    $\rho := \text{RefinePred}(\tau_c)$
  else
    $\rho := \text{true}$
  end if
end if
return $\langle t, \rho\rangle$

ExtendFrontier implementation

- Renaming
- InputConstraints
- ExitConstraints
- Graph construction
- Extract test input
- Extract refinement predicate

let $\langle S_{k-1}, \_\rangle = RS_{k-1}$
$\langle \phi, \mathcal{S} \rangle := \mathrm{ExecuteSymbolic}(\tau_c, P, \mathcal{P}, \mathcal{G})$
$op := \mathrm{Op}(S_{k-1}, S_k)$
if op **matches** $\mathsf{v} := \mathsf{f}(a_0, \ldots, a_n)$ **then**
$\quad$ **let** $\langle \rho_k, \_\rangle = S_k$
$\quad$ $P' := \mathrm{Lookup}(\mathsf{f}, \mathcal{P})$
$\quad$ $\pi := \mathrm{CreateVariableRenamer}(\mathrm{locals}(P) \cup \{v, v_0 \mid \forall v \in \mathrm{params}(P)\})$
$\quad$ $\phi_{ic} := \left(\bigwedge_{v_i \in \mathrm{params}(P')} v_i = \pi(a_i) \right) \wedge \left(\bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e) \right) \wedge \pi(\phi)$
$\quad$ $\phi_{ec} := \pi(\rho_k[@r/\mathsf{v}])$
$\quad$ $\mathcal{G}' := \mathrm{ReconstructGraphsAndInsertConstraints}(\mathcal{G}, \phi_{ic}, \phi_{ec}, P')$
$\quad$ $\langle r, z \rangle = \mathrm{DashLoop}(\mathcal{G}', \mathcal{P}, P')$
$\quad$ **if** $r = \textsc{fail}$ **then**
$\quad\quad$ $t := \pi^{-1}\left(z \setminus \{v_0 \mid \forall v \in \mathrm{params}(P')\} \right) \setminus \left(\mathrm{locals}(P) \cup \mathrm{params}(P) \right)$
$\quad\quad$ $\rho := \mathrm{true}$
$\quad$ **else**
$\quad\quad$ $t := \textsc{unsat}$
$\quad\quad$ $\rho := \pi^{-1}\left(\left(\bigvee_{\rho_i \in \mathrm{InitialRefines}(z)} \rho_i \right) [a_0/v_0, \ldots, a_n/v_n] \;\mid v_i \in \mathrm{params}(P') \right)$
$\quad$ **end if**
else
$\quad$ $t := \mathrm{IsSAT}(\phi, P)$
$\quad$ **if** $t = \textsc{unsat}$ **then**
$\quad\quad$ $\rho := \mathrm{RefinePred}(\tau_c)$
$\quad$ **else**
$\quad\quad$ $\rho := \mathrm{true}$
$\quad$ **end if**
end if
return $\langle t, \rho \rangle$

Scaling to large programs

Does DASH scale to large programs?

Scaling to large programs

Does DASH scale to large programs?

- DASH is single threaded

Scaling to large programs

Does DASH scale to large programs?

- DASH is single threaded
- DASH does not cache prior results

Scaling to large programs

Does DASH scale to large programs?

- DASH is single threaded
- DASH does not cache prior results

How can we improve the situation?

Scaling to large programs

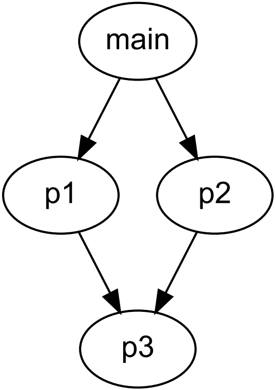
Does DASH scale to large programs?

- DASH is single threaded
- DASH does not cache prior results

How can we improve the situation?
The article *Parallelizing Top-Down Interprocedural Analysis* presents BOLT:

- Top-Down Interprocedural analysis
- Uses summaries to avoid recomputation
- Is modular $\mapsto$ multithreading support

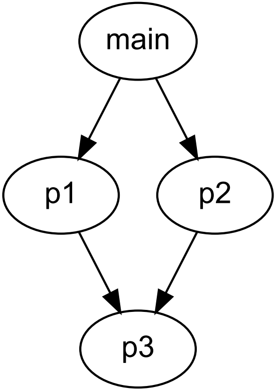
Call graph



- Each procedure is a node
- Edges correspond to procedure calls

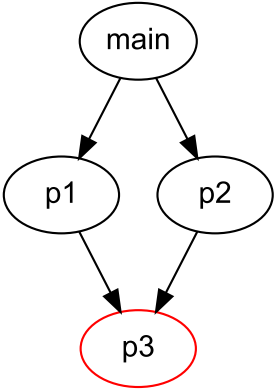
Bottom-up interprocedural analysis

- Analyzes leafs first, assuming any input can be given



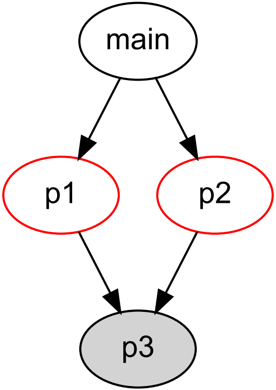
Bottom-up interprocedural analysis

- Analyzes leafs first, assuming any input can be given
  - $p3$ analyzed first



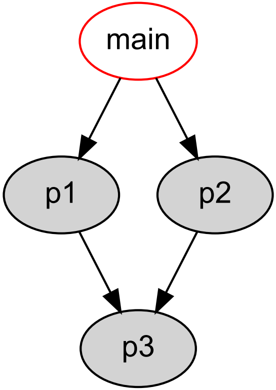
Bottom-up interprocedural analysis

- Analyzes leafs first, assuming any input can be given
  - *p*3 analyzed first
  - Summery for *p*3 can be used to analyze *p*1 and *p*2



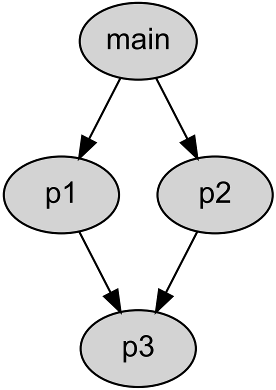
Bottom-up interprocedural analysis

- Analyzes leafs first, assuming any input can be given
  - *p*3 analyzed first
  - Summery for *p*3 can be used to analyze *p*1 and *p*2
  - Summaries for *p*1 and *p*2 can be used to analyze *main*



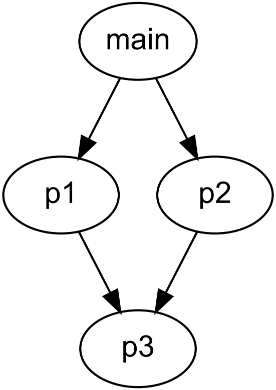
Bottom-up interprocedural analysis

- Analyzes leafs first, assuming any input can be given
  - $p3$ analyzed first
  - Summery for $p3$ can be used to analyze $p1$ and $p2$
  - Summaries for $p1$ and $p2$ can be used to analyze *main*
- Callers of a procedure p_i is decoupled from the analysis of the body of p_i $\mapsto$ easily parallelizable



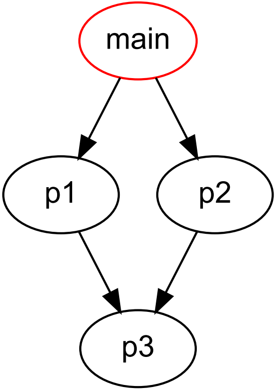
Top-down interprocedural analysis

- Analyzes main procedure first.
- Analyze procedures in called context



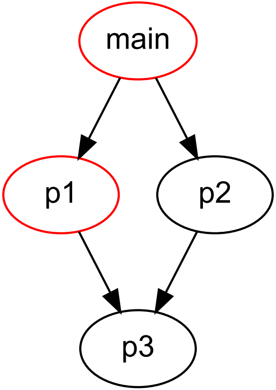
Top-down interprocedural analysis

- Analyzes main procedure first.
- Analyze procedures in called context
  - analyze *main* procedure



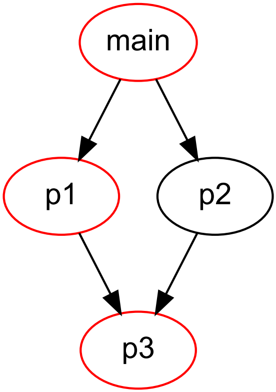
Top-down interprocedural analysis

- Analyzes main procedure first.
- Analyze procedures in called context
  - analyze *main* procedure
  - $main \mapsto p1$



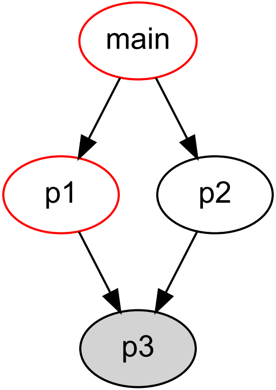
Top-down interprocedural analysis

- Analyzes main procedure first.
- Analyze procedures in called context
  - analyze *main* procedure
  - $main \mapsto p1$
  - $p1 \mapsto p3$



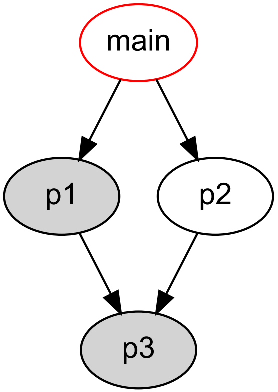
Top-down interprocedural analysis

- Analyzes main procedure first.
- Analyze procedures in called context
  - analyze *main* procedure
  - $main \mapsto p1$
  - $p1 \mapsto p3$
  - Summary generated for $p3$



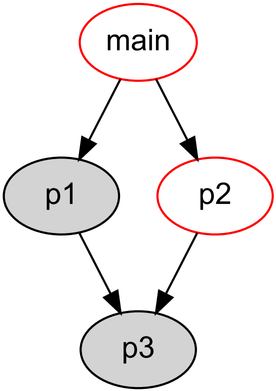
Top-down interprocedural analysis

- Analyzes main procedure first.
- Analyze procedures in called context
  - analyze *main* procedure
  - $main \mapsto p1$
  - $p1 \mapsto p3$
  - Summary generated for $p3$
  - Summary generated for $p1$



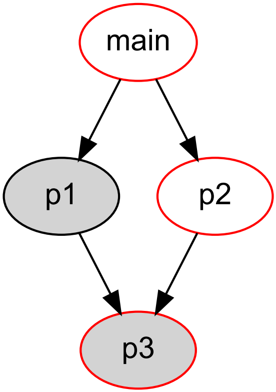
Top-down interprocedural analysis

- Analyzes main procedure first.
- Analyze procedures in called context
  - analyze *main* procedure
  - $main \mapsto p1$
  - $p1 \mapsto p3$
  - Summary generated for $p3$
  - Summary generated for $p1$
  - $main \mapsto p2$



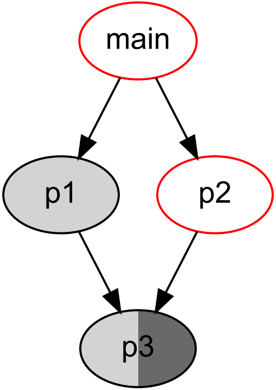
Top-down interprocedural analysis

- Analyzes main procedure first.
- Analyze procedures in called context
  - analyze *main* procedure
  - $main \mapsto p1$
  - $p1 \mapsto p3$
  - Summary generated for $p3$
  - Summary generated for $p1$
  - $main \mapsto p2$
  - $p2 \mapsto p3$



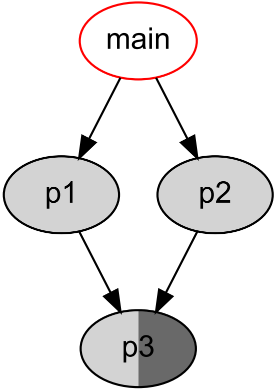
Top-down interprocedural analysis

- Analyzes main procedure first.
- Analyze procedures in called context
  - analyze *main* procedure
  - $main \mapsto p1$
  - $p1 \mapsto p3$
  - Summary generated for $p3$
  - Summary generated for $p1$
  - $main \mapsto p2$
  - $p2 \mapsto p3$
  - New summary generated for $p3$



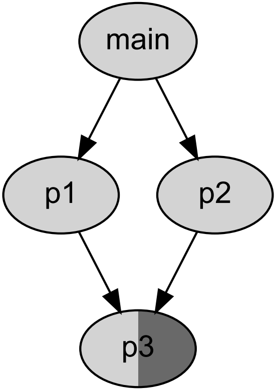
Top-down interprocedural analysis

- Analyzes main procedure first.
- Analyze procedures in called context
  - analyze *main* procedure
  - $main \mapsto p1$
  - $p1 \mapsto p3$
  - Summary generated for $p3$
  - Summary generated for $p1$
  - $main \mapsto p2$
  - $p2 \mapsto p3$
  - New summary generated for $p3$
  - Summary generated for $p2$



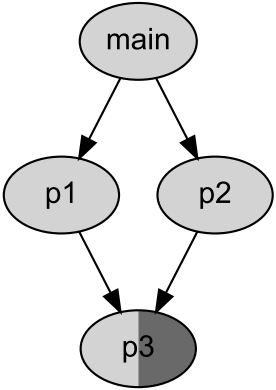
Top-down interprocedural analysis

- Analyzes main procedure first.
- Analyze procedures in called context
  - analyze *main* procedure
  - $main \mapsto p1$
  - $p1 \mapsto p3$
  - Summary generated for $p3$
  - Summary generated for $p1$
  - $main \mapsto p2$
  - $p2 \mapsto p3$
  - New summary generated for $p3$
  - Summary generated for $p2$
  - Analysis ends in *main*

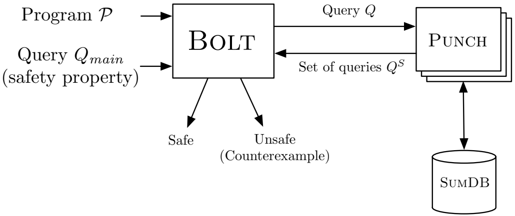


Top-down interprocedural analysis

- Analyzes main procedure first.
- Analyze procedures in called context
  - analyze *main* procedure
  - $main \mapsto p1$
  - $p1 \mapsto p3$
  - Summary generated for $p3$
  - Summary generated for $p1$
  - $main \mapsto p2$
  - $p2 \mapsto p3$
  - New summary generated for $p3$
  - Summary generated for $p2$
  - Analysis ends in *main*
- Fine grained dependencies $\mapsto$ difficult to parallelize



BOLT



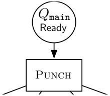
- BOLT calls PUNCH with a query Q about a procedure P_i
- PUNCH analyzes a single procedure.
- Checks if SUMDB contains a summery that answers the query
- If not, it returns a set Q^s to bolt containing the unanswered query
- If all answers are found, it answers the query Q

BOLT example

```
int foo(int p_foo);
int bar(int p_bar);
int baz(int p_baz);

main(int i, int j){
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (j > -10)
    x = bar(i);
  else
    x = baz(j);

  y = x + 5;
  assert(y > 0);
}
```

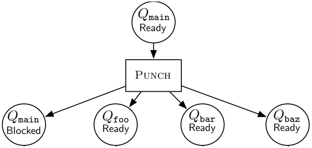
BOLT example

```
int foo(int p_foo);
int bar(int p_bar);
int baz(int p_baz);

main(int i, int j){
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (j > -10)
    x = bar(i);
  else
    x = baz(j);

  y = x + 5;
  assert(y > 0);
}
```



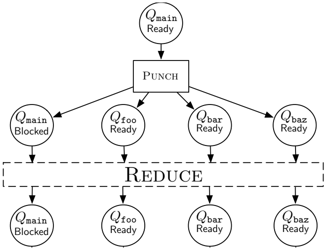
BOLT example

```
int foo(int p_foo);
int bar(int p_bar);
int baz(int p_baz);

main(int i, int j){
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (j > -10)
    x = bar(i);
  else
    x = baz(j);

  y = x + 5;
  assert(y > 0);
}
```



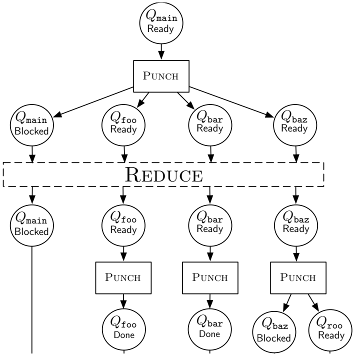
BOLT example

```
int foo(int p_foo);
int bar(int p_bar);
int baz(int p_baz);

main(int i, int j){
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (j > -10)
    x = bar(i);
  else
    x = baz(j);

  y = x + 5;
  assert(y > 0);
}
```



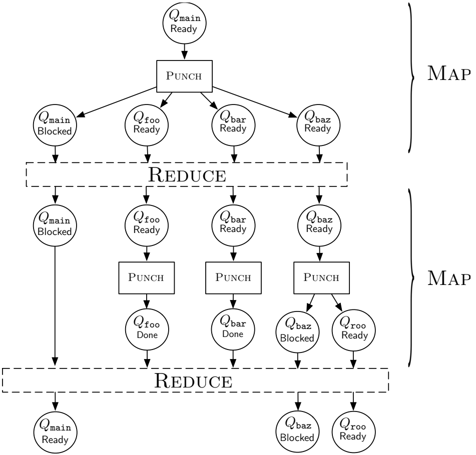
BOLT example

```
int foo(int p_foo);
int bar(int p_bar);
int baz(int p_baz);

main(int i, int j){
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (j > -10)
    x = bar(i);
  else
    x = baz(j);

  y = x + 5;
  assert(y > 0);
}
```



BOLT example

```
int foo(int p_foo);
int bar(int p_bar);
int baz(int p_baz);

main(int i, int j){
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (j > -10)
    x = bar(i);
  else
    x = baz(j);

  y = x + 5;
  assert(y > 0);
}
```



Summaries

- Two kinds of summaries:

Summaries

- Two kinds of summaries:
  - Must summary: $\langle \varphi_1 \overset{must}{\Longrightarrow}_{P_i} \varphi_2 \rangle$
    All states starting in state φ_1 in procedure P_i ends in a state φ_2

Summaries

- Two kinds of summaries:
  - Must summary: $\langle \varphi_1 \overset{must}{\Longrightarrow}_{P_i} \varphi_2 \rangle$
    All states starting in state φ_1 in procedure P_i ends in a state φ_2
  - Not-may summary: $\langle \varphi_1 \overset{\neg may}{\Longrightarrow}_{P_i} \varphi_2 \rangle$
    All states starting in state φ_1 in procedure P_i cannot reach a state φ_2

Queries and use of summaries

- Query: $\langle \varphi_1 \overset{?}{\Longrightarrow}_{P_i} \varphi_2 \rangle$
  Can procedure P_i starting in state φ_1 reach a state φ_2?

Queries and use of summaries

- Query: $\langle \varphi_1 \stackrel{?}{\Longrightarrow}_{P_i} \varphi_2 \rangle$
  Can procedure P_i starting in state φ_1 reach a state φ_2?
  In DASH: $\varphi_1 = \text{InputConstraints}$ and $\varphi_2 = S_k$

Queries and use of summaries

- Query: $\langle \varphi_1 \stackrel{?}{\Longrightarrow}_{P_i} \varphi_2 \rangle$
  Can procedure P_i starting in state φ_1 reach a state φ_2?
  In DASH: $\varphi_1 = \text{InputConstraints}$ and $\varphi_2 = S_k$
- **Yes** if: $S = \langle \hat{\varphi}_1 \stackrel{must}{\Longrightarrow}_{P_i} \hat{\varphi}_2 \rangle$ where $\hat{\varphi}_1 \subseteq \varphi_1$ and $\varphi_2 \cap \hat{\varphi}_2 \neq \emptyset$

Queries and use of summaries

- Query: $\langle \varphi_1 \overset{?}{\Longrightarrow}_{P_i} \varphi_2 \rangle$
  Can procedure P_i starting in state φ_1 reach a state φ_2?
  In DASH: $\varphi_1 = \text{InputConstraints}$ and $\varphi_2 = S_k$
- **Yes** if: $S = \langle \hat{\varphi}_1 \overset{must}{\Longrightarrow}_{P_i} \hat{\varphi}_2 \rangle$ where $\hat{\varphi}_1 \subseteq \varphi_1$ and $\varphi_2 \cap \hat{\varphi}_2 \neq \emptyset$
- **No** if: $S = \langle \hat{\varphi}_1 \overset{\neg may}{\Longrightarrow}_{P_i} \hat{\varphi}_2 \rangle$ where $\varphi_1 \subseteq \hat{\varphi}_1$ and $\varphi_2 \subseteq \hat{\varphi}_2$

Experimantal results

- Implemented DASH in BOLT
  - Single threaded analysis: 26 hours
  - Parallel analysis: 7 hours

Experimantal results

- Implemented DASH in BOLT
  - Single threaded analysis: 26 hours
  - Parallel analysis: 7 hours
  - Average speedup: 3.71x