Karlsruhe Institute of Technology

Refinement of Path Conditions for Information Flow Analysis

Bachelor Thesis of

Stephan Gocht

At the Department of Informatics
Institute for Theoretical Computer Science (ITI)
Automated Software Analysis (ASA)

Reviewer:	JProf. Dr. Mana Taghdiri
Second reviewer:	Dr. Carsten Sinz

Duration: 06. August 2014 – 05. December 2014

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, December 2, 2014

. . . .

(**Stephan Gocht**)

Abstract

Path conditions are a static analysis tool for information flow control (IFC). They can be used to produce witnesses for an illegal flow, which do not necessarily represent a concrete execution of the program. This bachelor thesis will provide a detailed approach to eliminate these false witnesses using counterexample guided abstraction refinement (CEGAR) and thereby increase precision.

As not all values satisfying the PC need to occur simultaneously during a program execution, a property is introduced which is true iff the values occur during the program execution. Some values are always occurring simultaneously if a flow exists. This information can be used to increase precision and is added to the described property, without using temporal logic. Finally, the CEGAR approach is adopted to provide an algorithm for checking this property.

Contents

1. Introduction

Information flow control (IFC) is used to ensure an information flow policy. The most basic information flow policy is non-interference introduced in [5]. It states that one realm may not influence another realm. For example, we can divide the input parameters as well as the output of a method into high and low security. If non-interference holds, no information from the high input may reach the low output. There are a lot of approaches for information flow control. An overview can be found in [6].

We will focus on path conditions (PC). They are based on program dependence graphs (PDG), which show the influence of statements to the control flow and also where information is used, which was stored by a statement. A path condition is a formula over program variables which describes a necessary condition for a flow. As it is not sufficient, a solution only indicates a potential flow. There are false positive results.

The solution for a PC assigns values to variables in the PC, which can be used as a witness for a potential flow. This witness can then be verified. But the witnesses of path conditions as described in [12, 14] do not necessarily conform to an execution of the program. [15] describes a novel approach of using counter example guided abstraction refinement (CEGAR) to increase the precision of path conditions, by ensuring that only solutions are given, which conform to an actual execution of the program.

We will show how this can be done in detail and prove that the resulting values conform to a program execution. Therefore, we will need to reduce the PC as it can constrain variables too much, so that they cannot conform to a program execution. Using this modified PC we will define a property that describes the meaning of a conform solution. We will then follow the CEGAR approach as in [15], by introducing a more formal definition of a valid execution and the refinement. Additionally, we will show that these definitions will result in the defined property. On the way we will also notice that we can add temporal information to the PC, such that the conforming solution respects these constraints without using temporal logic. This is increasing the precision of the PC.

At the end, we will have a solution of the path condition, which conforms to an execution. But this does not mean that we will get rid of false witnesses regarding information flow. Nonetheless, we will have less false positive results as the experiments show.

A prototype implementation of the algorithm was used to perform some experiments. While the approach works well for small examples it lags scalability, but potential optimisations are identified.

2. Basic Concepts

We need some existing transformations and analysis methods for the program. This section gives the definitions used as well as an ongoing example for explanation.

2.1 Graph Basics

A graph is a tuple $G = (V, E)$ with a set of vertices V and a set of edges $E \subset V \times V$. A path $p \in V^{(+)}$ is a list of vertices $p = (p_0, p_1, ..., p_n)$, such that for all $i \in \{0, \dots, n-1\}$: $(p_i, p_{i+1}) \in E$. $|p|$ is the number of vertices in p so $|p| - 1$ is the length of a path. For $v \in V$ $|p|_v$ will be the number of times v occurs in p. Furthermore, we will use $v \in_V p$ for v being any vertex in p and $e \in_E p$ for e being any edge in p. $a \rightarrow^* b$ is a path from a to b and $a \rightarrow b$ is a path from a to b with length one. For $p = a \rightarrow^* b$ and $q = b \rightarrow^* c$, $p \circ q$ is the concatenation of path p and q. $\text{subpath}(p)$ is the set of sub paths of p and $\text{prefix}(p) = \{p' \mid \exists q : p' \circ q = p\}$ is the set of prefixes of p.

2.2 Static Single-Assignment Form

Static single assignment form (SSA) is an intermediate code representation, where each variable is only assigned in one statement. Each assignment uses subscript enumeration of variables to achieve this. If two different definitions reach the same statement, then phi statements are inserted. For a construction of the SSA form, please refer to [3].

We will only use SSA variables from now on, so if we talk about a variable we mean an SSA variable.

In figure 2.1 line 10 and 14 assign the variable sum, so in the SSA form both have different subscripts. Line 15 shows a Phi statement.

2.3 Control Flow Graph

The control flow graph (CFG) is a graph that represents the code. Nodes are statements and an edge between two nodes x, y indicates that the statement of y can be executed directly after the statement represented by x. The granularity of the nodes can vary, for example, one node per line or one node per assembler operation. We will use one line per statement, for details, see section B.2. For the CFG we use the definition from [4]:

Definition 1. *A control flow graph is a directed graph $G = (N, E)$ augmented with a unique entry node "start" and a unique exit node "end" such that each node in the graph has at most two successors. We assume that nodes with two successors have attributes "T" (true) and "F" (false) associated with the outgoing edges in the usual way. We further assume that for any node n in G there exists a path from start to n and a path from n to end.*

We will always use the control flow graph as representation of the code in SSA-form. This allows us to show different properties.

In figure 2.1c you can see the CFG of the sum example. There is a node labelled $i(\cdot)$ for each input parameter. Nodes for Phi statements of the SSA-form have a dashed border. Edges labelled "F" are blue and dashed, all other edges are black and dotted. The nodes are labelled with two numbers, the first number is the line number of the related statement. As some lines contain more than one statement, for example line 12 in figure 2.1b, we also use the second number to uniquely identify each node. For the meaning of the second number see appendix B.

Definition 2. *For each edge $e = (x, y) \in E$ we define the function $\hat{c}(e)$. If x has two ancestors $\hat{c}(e)$ is the condition for taking this edge. It is true otherwise.*

For example in the CFG in figure 2.1c $\hat{c}(13/27, 14/37) \equiv sum_{54} = 0$, while $\hat{c}(13/27, 16/52) \equiv sum_{54} \neq 0$.

2.3.1 Loops

As in standard compiler text books (p.a. [11] p. 191-197) we will use the concepts of natural loops, loop heads and reducibility.

Given a control flow graph $G_{CFG} = (V, E)$. Let $x, y \in V$. Then x dominates y iff every path from start to y contains x. A back edge is an edge $(x, y) \in E$ such that x is dominated by y. y is called loop head and x is called tail. The nodes of the natural loop are given by $\text{NatLoop}(x, y) = \{v \in V \mid \exists p = v \rightarrow^* x \land y \notin_V p \lor y = v\}$.

Additionally, for $v \in V$ we will define $\text{tails}(v) = \{x \in V \mid \exists y \in V : (x, y) \in E$ is a back edge $\land$ $v \in \text{NatLoop}(x, y) \lor x = \text{end}\}$

In figure 2.1 there is only one back edge: $e = (16/39, 12/54)$. Notice that the while node 12/45 is not the head of the loop, as the phi nodes are always executed before it. $\text{NatLoop}(e) = \{12/54, 12/56, 12/45, 13/27, 14/37, 16/52, 16/38\}$. For all of these nodes $\text{tails}(\cdot)$ is $\{16/39, end\}$.

G_{CFG} is reducible iff $E = E_F \cup E_B$ such that E_F and E_B are disjunct and (V, E_F) is a directed acyclic graph and $\forall (t, h) \in E_B : (t, h)$ is a back edge.

2.4 Program Dependence Graph

The program dependence graph is based on the control flow graph. It contains control dependencies that show if a node influences the execution of another node and data dependencies which show where a defined variable is used.

We use again the definition of [4]:

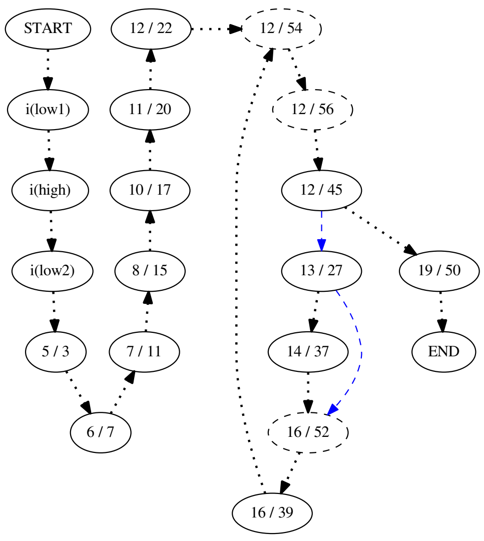
Definition 3. *A node x is post-dominated by a node y in G_{CFG} if every directed path from x to stop (not including x) contains y.*

```
5   int[] a = new int[3];
6   a[0] = low1;
7   a[1] = high;
8   a[2] = low2;
9
10  int sum = 0;
11  int i = 0;
12  while (i < 3) {
13      if (sum == 0) {
14          sum = sum + a[i];
15      }
16      i++;
17  }
18
19  return sum;
```

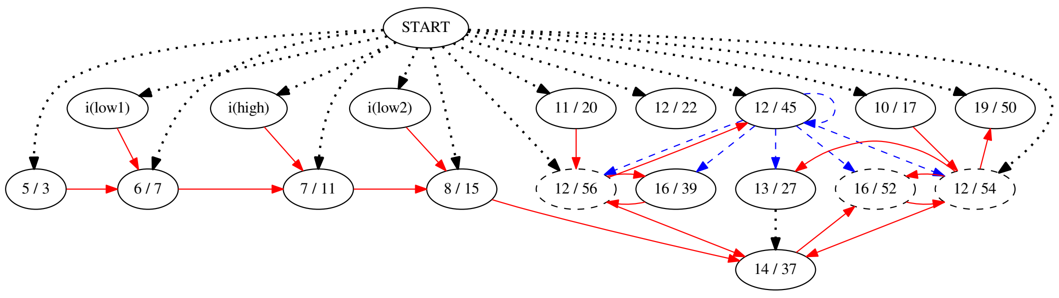
(a) Source Code

```
5   a_3 = new int[3];
6   a_7[0] = low1;
7   a_11[1] = high;
8   a_15[2] = low2;
9
10  sum_17 = 0;
11  i_20 = 0;
12  while (i_56 = Φ(i_20, i_39), sum_54 = Φ(sum_52,
        sum_17); i_56 < 3) {
13      if (sum_54 == 0) {
14          sum_37 = sum_54 + a_15[i_56];
15      } sum_52 = Φ(sum_54, sum_37);
16      i_39 = i_56 + 1;
17  }
18
19  return sum_54;
```

(b) Source Code in SSA Form



(c) Control Flow Graph



(d) Program Dependence Graph

Figure 2.1: Example Sum, from [15]

Definition 4. *Let G_{CFG} be a control flow graph. Let x and y be nodes in G_{CFG}. y is control dependent on x iff*

- *there exists a directed path p from x to y with any $z \in_V p$ (excluding x and y) post-dominated by y*
- *x is not post-dominated by y*

In difference to [4] our CFG is based on the SSA form, therefore we will use another definition of data dependence:

Definition 5. *Let G_{CFG} be a control flow graph. Let x and y be nodes in G_{CFG}. y is data dependent on x iff x defines a variable used in y.*

Note: A definition of an array field is considered to use the previously defined array, so array definitions are treated as killing definitions.

Definition 6. *Let $G_{CFG} = (V, E)$ be a control flow graph. Let $E_{cd} = \{(x, y) \mid x, y \in V$ and y is control dependent on $x\}$ and $E_{dd} = \{(x, y) \mid x, y \in V$ and y is data dependent on $x\}$ The program dependence graph of G_{CFG} is a directed graph $G_{PDG} = (V', E')$ such that $V = V'$ and $E' = E_{cd} \cup E_{dd}$.*

Figure 2.1d shows the PDG for the CFG in 2.1c. Control dependencies are dashed blue or dotted black, while data dependencies are continuous and red.

For paths in E_{dd} we will write $x \to^*_{dd} y$ and for paths in E_{cd} $x \to^*_{cd} y$. For $(x, y) \in E_{cd}$ there is exactly one edge $(x, z) \in E$ such that z is post dominated by y or $z = y$. We define $c(x, y) := \hat{c}(x, z)$.

2.5 Path Conditions

Path conditions were originally introduced by [13] and are a necessary condition for a flow. More formally this means: Let $\alpha, \beta \in V$. A path condition is a formula F such that: There is an information flow from α to $\beta \Rightarrow$ there is a satisfying solution for F. They are built upon the theorem that a flow can only happen along paths in the program dependence graph:

Let $CH(\alpha, \beta) = \{p \mid p$ is an acyclic path in the program dependence graph from α to $\beta\}$.

$$PC(\alpha, \beta) \equiv \bigvee_{p \in CH(\alpha,\beta)} \Bigg(\underbrace{\Big(\bigwedge_{n \in_V p} E(n) \Big)}_{\text{execution condition}} \wedge \underbrace{\Big(\bigwedge_{d \in D(p)} \delta(d) \Big)}_{\text{data flow condition}} \Bigg) \wedge \underbrace{\Phi}_{\text{phi condition}}$$

In the presence of loop-carried dependencies some adaptations are necessary, which will not be described here. For those refer to [8, 9, 12, 14] instead.

2.5.1 Execution Condition

For the flow to happen along a specific path it is necessary that all statements of this path are executed. An execution condition is a necessary condition for a statement to be executed. Let $P(n) = \{p \mid p = start \to^*_{cd} n \wedge p$ is acyclic$\}$ Given a statements node n the execution condition is given by:

$$E(n) \equiv \bigvee_{p \in P(n)} \bigwedge_{e \in_E p} c(e).$$

2.5.2 Data Flow Condition

In the presence of arrays we can add further constraints using the index expressions. There is only a flow from a definition of an array value to a use if the indices are the same. Additionally the relevant array entry must not be overwritten. We use the same concept as in [14] for arrays. This requires that array definitions are treated as killing definitions in the program dependence graph. Now we will look for chains of data dependencies from the first definition to its use. In between, there can be further definitions, which are not allowed to have the same index for a flow to occur.

Given a path p in the PDG C_{PDG}. Let $D(p)$ be the set of all paths q such that:

- $q \in \text{subpath}(p)$
- $\forall e \in_E q : e \in E_{dd}$
- $\exists v \in \text{Var} : v$ is an array
- $\forall i \in \{0, \ldots, |q| - 2\} : v[x_i] \in \text{defs}(q_i)$
- $v[x_{|p|-1}] \in \text{uses}(q_{|q|-1})$
- q is the maximal sub path of p which satisfies these conditions

For $d \in D(p)$ we define $\delta(d) \equiv \bigwedge_{i \in \{1..|p|-2\}} x_0 \neq x_i \wedge x_0 = x_{|p|-1}$.

Notice: This definition requires that array values are not directly assigned to arrays. As 1: $a[i] = h$; 2: $a[k] = a[l]$; 3: return $a[m]$; would need to state $(i = m \wedge i \neq k) \vee (i = l \wedge m = k)$ but as it would be just one path, we would compute only the first part. This does not seem to be considered in [14]. However we can bypass this problem by introducing a temporal variable 1: $a[i] = h$; 2: $tmp = a[l]$; 3: $a[k] = tmp$; 4: return $a[m]$; now there are two paths (1,3,4) and (1,2,3,4) and the condition is computed correctly.

2.5.3 Phi Condition

In SSA form each variable is assigned only once and phi statements were introduced. A phi statement $v_i = \Phi(v_k, v_l)$ in the code can be represented as $v_i = v_k \vee v_i = v_l$. The phi condition contains all such conditions conjunctively joined. Additionally equations are added to relate different SSA forms of a variable along data dependence edges. Details can be found in [14].

2.5.4 Example

Assume we want to know if a flow can occur from the input parameter high to the return statement, in the program of figure 2.1. There is only one acyclic path $p = [i(\text{high})] \rightarrow_d^* [19/50] = (i(\text{high}), 7/11, 8/15, 14/37, 16/52, 12/54, 19/50)$. The execution conditions are true for all nodes but 14/37 and 16/52. $E(14/37) \equiv i_{56} < 3 \wedge sum_{54} = 0$, $E(16/52) \equiv i_{56} < 3$. The only proper sub path for a data flow condition is $d = (7/11, 8/15, 14/37)$ and $\delta(d) \equiv 1 \neq 2 \wedge 1 = i_{56} \equiv 1 = i_{56}$. And $\Phi \equiv (i_{56} = i_{20} \vee i_{56} = i_{39}) \wedge (sum_{54} = sum_{52} \vee sum_{54} = sum_{17}) \wedge (sum_{52} = sum_{54} \vee sum_{52} = sum_{37}) \wedge sum_{37} = sum_{54} \wedge i_{20} = 0 \wedge sum_{17} = 0$. For details about the phi condition, please refer to [15].

$$\begin{aligned}
PC(i(\text{high}), 19/50) \equiv & E(14/37) \wedge \delta(d) \wedge \Phi \\
\equiv & i_{56} < 3 \wedge sum_{54} = 0 \wedge 1 = i_{56} \wedge (i_{56} = i_{20} \vee i_{56} = i_{39}) \\
& \wedge (sum_{54} = sum_{52} \vee sum_{54} = sum_{17}) \\
& \wedge (sum_{52} = sum_{54} \vee sum_{52} = sum_{37}) \\
& \wedge sum_{37} = sum_{54} \wedge i_{20} = 0 \wedge sum_{17} = 0 \\
\equiv & i_{20} = 0 \wedge i_{39} = i_{56} = 1 \wedge sum_{17} = sum_{37} = sum_{52} = sum_{54} = 0
\end{aligned}$$

As this equation is satisfiable it indicates that there may be a flow, which is indeed the case, if the input parameters are not constrained further. If all input variables are asserted to be greater than zero ($PC(i(\text{high}), 19/50) \wedge \text{low1} > 0 \wedge \text{low2} > 0 \wedge \text{high} > 0$) then the formula is still satisfiable but no flow exists.

3. Environment

We need some definitions, lemmas and assumptions, which are provided in this chapter.

3.1 Formalized Program Execution

Let Var be the set of SSA-variables of the program. Values of input parameters are represented by a substitution i, which substitutes each input parameter with a term that does not contain free variables. Let $\mathcal{I}$ be the set of such substitutions. Let $G_{CFG} = (V, E)$ be the control flow graph of a program.

- A trace t is a path in G_{CFG} starting at the start node.
- A trace t is complete if t ends at the end node.
- For t trace and $v \in \text{Var}$ we define $\text{val}(t, v)$ as the last symbolic value assigned to v in t or $\perp$ if v was not assigned yet. The symbolic value only contains input parameters as free variables.
- $\text{val}(t, v, i) = i(\text{val}(t, v))$
- val_t is a substitution such that, any $v \in \text{Var} : val(t, v) \neq \perp$ is substituted by $\text{val}(t, v)$
- $\text{val}_{t,i}$ is a substitution such that, any $v \in \text{Var} : val(t, v) \neq \perp$ is substituted by $i(\text{val}(t, v))$
- $\text{guard}(t) = \bigwedge_{t' \circ (x \to y) \in \text{prefix}(t)} \text{val}_{t'}(\hat{c}(x, y))$
- $\text{guard}(t)_{|A} = \bigwedge_{t' \circ (x \to y) \in \text{prefix}(t) : x \in A} \text{val}_{t'}(\hat{c}(x, y))$
- An execution is a tuple $\xi = (t, i)$ where t is a complete trace and $i \in \mathcal{I}$ and $i(\text{guard}(t)) \equiv \text{true}$ (i.e. the path t corresponds to an execution of the program with these input values).

Again, we will use the example from figure 2.1 for clarification: Let p be the acyclic path from start to 12/45. $val(p, i_{56}) = 0$, it does not depend on the input variables. $val(p, a_{15}[1]) = \text{high}$, given the input $i = \{\text{low1}/0, \text{high}/1, \text{low2}/2\}$ $val(p, a_{15}[1], i) = i(val(p, a_{15}[1])) = i(high) = 1$. $\hat{c}(12/45, 19/50) = i_{56} \geq 3$, $val_p(\hat{c}(12/45, 19/50)) = 0 \geq 3 = \text{false}$. So $p \circ (12/45 \to 19/50)$ is never part of an execution, as a guard of a trace containing $p \circ (12/45 \to 19/50)$ would evaluate to false as well.

Lemma 1. *Let $\xi = (t, i), \xi' = (t', i')$ be executions. If $i(guard(t')) \equiv TRUE \Rightarrow t = t'$*

Proof. Assume $t \neq t'$. Let $p \in \text{prefix}(t) \cap \text{prefix}(t')$ be the longest path in this set and $(a, b) \in E$ such that $p \circ (a \to b) \in \text{prefix}(t)$ and (a, c) such that $p \circ (a \to c) \in \text{prefix}(t')$. $b \neq c$ by construction and $\hat{c}(a, b) = \neg\hat{c}(a, c)$. $i(val_p(\hat{c}(a, b))) \equiv TRUE$ as ξ is an execution. This means false $\equiv i(\text{val}_p(\neg\hat{c}(a, b))) \equiv i(\text{val}_p(\hat{c}(a, c)))$, but $\text{val}_p(\hat{c}(a, c))$ is part of $guard(t')$ so this is a contradiction to $i(guard(t')) \equiv TRUE$ □

Lemma 2. *Let (t, h) be a back edge and $x \in NatLoop(t, h)$ then h dominates x.*

Proof. $x \in \text{NatLoop}(t, h) \Rightarrow \exists p = (x \to^* t)$ as h dominates t it have to be h dominates x as well or $\exists q = \text{start} \to^* t$ such that $h \notin_V q$ as $h \notin_V p$. Which is a contradiction to h dominates t. □

3.2 Set-Up

From now on we will have a program in its static single assignment form with its control flow graph $G_{CFG} = (V, E)$ and the resulting program dependence graph $G_{PDG} = (N, E_d)$ where $E_d = E_{cd} \cup E_{dd}$. Furthermore, we will have two nodes $\alpha, \beta \in N$ and are interested if a flow from α to β can occur.

Assumption 1. *G_{CFG} is reducible.*

We will let $V_{|\neg\Phi} \subset V$ be the set of nodes which are not phi nodes.

Assumption 2. *Each node $n \in V_{|\neg\Phi}$ has a unique acyclic path $p = start \to^*_{cd} n$ in G_{PDG}.*

Assumption 3. *If $v \in Var$ is an arrays than we assume $\forall n \in N : v \in defs(n) \to v \notin uses(n)$*

Assumption 1 and 2 are covered by structured programs. By [1] all programs can be represented as such. Assumption 3 can be reached by introducing temporal variables. So this is not a limitation in a theoretical sense, as all programs can be transformed to conform the assumptions.

4. Path-Conditions and Variable-Values

The path conditions described in section 2.5 are a necessary condition for a flow. We want a solution that satisfies this necessary condition and represents an execution of the program. So the solution should conform to an execution. Actually, that is not possible with the standard path conditions as solutions can be misleading witnesses as described in [9]: It can happen that the path condition implies a value that may never occur during a program run. For example in section 2.5.4 we constructed and simplified the path condition for figure 2.1. Simplification leads to $\text{sum}_{37} = 0$, but for the input $\text{low1} = 0$ $\text{low2} = \text{high} = 1$, there is never a state of the program where sum_{37} takes the value 0.

What we need is a condition which does not have this behaviour, but preserves the basic ideas of path conditions. Additionally, we need to consider, where and when the values need to occur during the execution, so their occurrence do relate to the condition we are checking.

4.1 Execution Conditions

Let us review the execution condition for a node n. It constrains the variables at branching nodes, such that the constraint is necessary for the execution of n. So the variables at an execution of such a branching node satisfies the branching condition ($c(\cdot)$), which leads to the execution of n. Due to the SSA form these values will not change until the execution of n. So what we can prove is:

Theorem 1. *Let $G_{CFg} = (V, E)$ be any CFG and G_{PDG} its PDG. Let $n \in V_{|\neg\Phi}$ and $\xi = (t, i)$ execution. Let $p \in prefix(t)$ be a path ending at an occurrence of n. Then $val_{p,i}(E(n)) \equiv true$.*

That we are only talking about nodes in $V_{|\neg\Phi}$, non phi nodes, has technical reasons: As we can see in figure 2.1 the phi nodes (12/54, 12/56) before the while node (12/45) have two ancestors in the program dependence graph. So we need to consider more than one control dependence path from start to the node. But we do not need to care about the execution of phi nodes along the analysed path of the PDG as SSA form guarantees its execution.

Before theorem 1 is proven some lemmas are needed:

Lemma 3. *Let $x, y \in V_{|\neg\Phi}, x \neq y, y$ control dependent on x, $p = START \rightarrow^* y$, $|p|_y = 1$. Let $q = getControlDependentPath(p)$ be the sub path of p constructed by using the algorithm in figure 4.1. Then $q = x \rightarrow^* y$.*

```
procedure getControlDependentPath($p = start \rightarrow^+ y$)
    $i \leftarrow |p| - 2$
    while $p_i$ is post dominated by $y$ do
        $i \leftarrow i - 1$
    end while
    $q \leftarrow (p_i, p_{i+1}, \dots, p_{|p|-1})$
                        ▷ $p_{|p|-1} = y$, $p_{i+1}, \dots, p_{|p|-2}$ post dom by y, $p_i$ not post dom by $y$
    return $q$
end procedure
```

Figure 4.1: Retrieving a control dependence showing path

Proof. By construction q is of the form $q = z \rightarrow^* y$ such that y is control dependent on z. As $y \in V_{|\neg\Phi}$ and $x \neq y$ it is $z = x$, by assumption 2. □

Lemma 4. *Given a CFG $G_{CFG} = (V, E)$. Let $x, y \in V_{|\neg\Phi}$, $x \neq y$ such that y is control dependent on x, and an execution $\xi = (t, i)$ such that $y \in_V t$ (y is executed). Let $p \in prefix(t)$ such that p ends at y and let $q = getControlDependentPath(p)$ and p' such that*

$$p = \underbrace{(START \rightarrow^* x)}_{=p'} \circ \underbrace{(x \rightarrow^* y)}_{=q}$$

Then $val_{p',i}(c(x \rightarrow_{cd} y)) \equiv true$.

Proof. As q_1 is either y or post dominated by y it is $\hat{c}(x, q_1) = c(x, y)$ by definition of $c(\cdot)$. As p' is based on a valid execution, it is: true $\equiv \text{val}_{p',i}(\hat{c}(x, q_1)) \equiv \text{val}_{p',i}(c(x, y))$. □

Theorem 1. *Let $G_{CFg} = (V, E)$ be any CFG and G_{PDG} its PDG. Let $n \in V_{|\neg\Phi}$ and $\xi = (t, i)$ execution. Let $p \in prefix(t)$ be a path ending at an occurrence of n. Then $val_{p,i}(E(n)) \equiv true$.*

Proof. Let $\pi = \text{start} \rightarrow^*_{cd} n$ be an acyclic path in G_{PDG}. π is uniquely defined by assertion 2. Let $k = |\pi| - 1$ $\mathbb{G}_k = \{0, \dots, k-1\}$. Now we break down p into k parts such that each part relates to one edge in π:

- $p^0 \circ \dots \circ p^{k-1} = p$
- $\forall l \in \mathbb{G}_k : p^l = (\pi_l \rightarrow^* \pi_{l+1})$
- $\forall l \in \mathbb{G}_k : \left|p^l\right|_{\pi_l} = 1$
- $\forall l, e \in \mathbb{G}_k; l + 1 < e : \pi_{l+1} \notin p^e$

Such a segmentation can always be achieved by using the algorithm from figure 4.1 repeatedly.

For $l \in \mathbb{G}_k$ $t^l := p^0 \circ \dots \circ p^l$ is the trace until the execution of π_{l+1}. From lemma 4 we know $\forall l \in \{0, \dots, |p| - 2\} : \text{val}_{t^l,i}(c(\pi_{l+1}, \pi_{l+2})) \equiv \text{true}$. Therefore, it is enough to show that the values of the variables do not change from the point of use in $c(\cdot)$ to the execution of n: Assume $\exists l \in \mathbb{G}_k \exists v \in \text{Var} : \text{val}(t^l, v, i) \neq \text{val}(p, v, i)$. Let s be the unique node, which assigns that variable v. It exists $e \in \mathbb{G}_k; l \leq e$ such that $s \in p^e$, or the value of the variable could not have changed since the execution of π_l to the execution of n. By construction of the SSA-form we know s dominates π_l and therefore $\exists q = \text{start} \rightarrow^* s : \pi_l \notin q$. And as $s \in p^e$ there is a path $q' = s \rightarrow^* \pi_{e+1}$ and $\pi_l \notin q'$ as $\pi_l \notin p^e$ by construction. $\pi_l \notin q \circ q' = \text{start} \rightarrow^* \pi_{e+1}$. It is a contradiction that such a path exists by using lemma 3 inductively.

We now know that $\forall l \in \mathbb{G}_k : \text{val}_{p,i}(c(\pi_{l+1} \rightarrow_{cd} \pi_{l+2})) \equiv \text{true}$ and therefore we have $\text{true} \equiv \bigwedge_{l \in \mathbb{G}_k} \text{val}_{p,i}(c(\pi_l, \pi_{l+1})) \equiv \text{val}_{t,i}(E(n))$. □

4.2 Coeval Conditions

Theorem 1 does not only state that values of the program do satisfy the execution condition, but also provides a necessary condition for the execution of a node: There has to be a single point during the execution where the values of that point do satisfy the execution condition.

Using this information, we can make the PC more precise. For example, in figure 4.2 we can see that it is not sufficient that a combination of occurred values satisfies the execution condition: $E(16) = (e_{38} = 1) \wedge (i_{40} = 0) \wedge (i_{40} < 2)$. During an execution i_{40} will be zero and e_{38} will be one, but line 16 will never be executed. The variables i_{40} and e_{38} do not take the necessary values during the same loop iteration.

If we have more than one execution condition, then we do also need more than a single point during an execution to fulfill the conditions: In the example of figure 4.3 the path condition is $E(17) \wedge E(20) = (i_{51} = 0) \wedge (e_{49} = 1) \wedge (i_{51} < 2)$. There is not a single point of execution, which satisfies the condition, as $i_{51} = e_{49} = 0$ during the first iteration and $i_{51} = e_{49} = 1$ during the second iteration. But both nodes are executed in every execution.

We see that sometimes specific values of variables have to occur together and sometimes they do not. To distinguish these cases we will tag each variable with an additional superscript. If two variables have the same superscript it means they have to occur together:

Definition 7.

$$\hat{E}(n) :\equiv \begin{cases} \bigwedge_{e \in cdPath(n)} tag_{idx(n)}(c(e)) & \textit{for } n \in V_{|\neg\Phi} \\ \textit{true} & \textit{otherwise} \end{cases}$$

where $cdPath(n) = start \rightarrow^*_{cd} n$ is the unique acyclic control dependence path to n and $tag_k(T)$ is a substitution, which replaces each free variable in T with the same variable with k as its superscript and idx returns a unique number for each node.

Note, that due to assumption 2 we have only one path from start to n left that we need to consider.

For example 4.2 we get

$$\hat{E}(16) \equiv (e^{19}_{38} = 1) \wedge (i^{19}_{40} = 0) \wedge (i^{19}_{40} < 2)$$

This can be interpreted as there is a single point during execution where all three equations hold. For example 4.3 we get

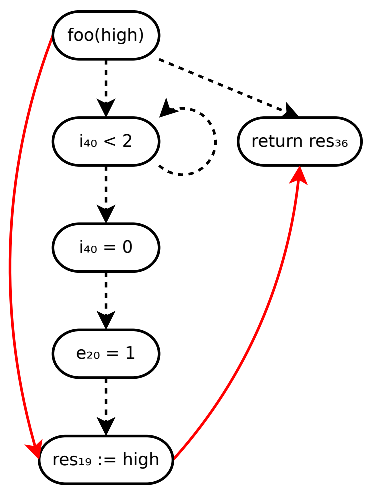
$$\hat{E}(17) \wedge \hat{E}(20) \equiv (i^{18}_{51} = 0) \wedge (i^{18}_{51} < 2) \wedge (e^{25}_{49} = 1) \wedge (i^{25}_{51} < 2)$$

This can be interpreted as there is one point during execution where the first two equations hold and another point where the last two equations hold.

The superscript ensures that the execution conditions for two different nodes do never contradict each other.

```
int foo(int high) {
    res_1 = 0;
    e_3 = 0;
    i_5 = 0;

    while (i_40 = Φ(i_5, i_23),
            e_38 = Φ(e_20, e_3),
            res_36 = Φ(res_34, res_1),
        i_40 < 2) {

        if (i_40 == 0)
            if (e_38 == 1)
                res_19 = high;
        res_34 = Φ(res_19, res_36);
        e_20 = e_38 + 1;
        i_23 = i_40 + 1;
    }
    return res_36;
}
```

(a) Source Code

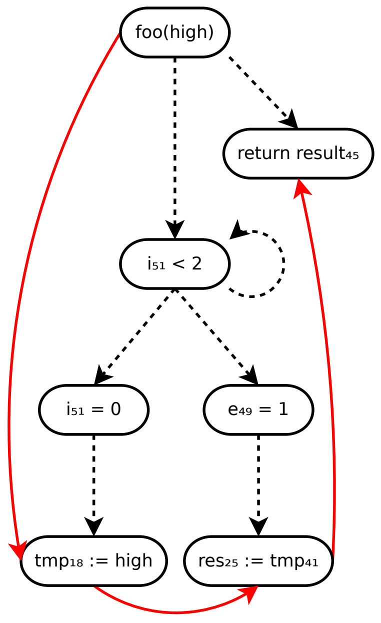


(b) Shematic PDG

Figure 4.2: Example of coeval conditions

```
int foo(int high) {
    res_1 = 0;
    tmp_3 = 0;
    e_5 = 0;
    i_7 = 0;

    while (i_51 = Φ(i_7, i_29),
            e_49 = Φ(e_5, e_26),
            tmp_47 = Φ(tmp_41, tmp_3),
            res_45 = Φ(res_43, res_1),
        (i_51 < 2) {

        if (i_51 == 0)
            tmp_18 = high;
        tmp_41 = Φ(tmp_18, tmp_47);
        if (e_49 == 1)
            res_25 = tmp_41;
        res_43 = Φ(res_25, res_45);
        e_26 = e_49 + 1;
        i_29 = i_51 + 1;
    }

    return res_45;
}
```

(a) Source Code



(b) Shematic PDG

Figure 4.3: Example of non coeval conditions

4.3 Data Flow Conditions

We need to adopt the data flow conditions to the new concept as well. As they constrain variables at the very execution of a specific node we just need to add the correct tag:

Definition 8. *Let $d = a \rightarrow^*_{dd} b$ such that $a[y] \in uses(b)$ and $\forall l \in \{0..|d|-2\} : a[x_l] \in defs(d_l)$.*

$$\begin{aligned}\hat{\delta}(d) \equiv \bigwedge_{l \in \{1..|p|-2\}} & tag_{idx(a)}(x_0) \neq tag_{idx(d_l)}(x_l) \\ & \wedge\, tag_{idx(a)}(x_0) = tag_{idx(b)}(y)\end{aligned}$$

4.4 Phi Conditions

Phi conditions were meant to relate the SSA variables, as they have to stay the same from definition to use, if no loop-carried dependencies interfere. As we will ensure that only values are used for a witness which also occur during the execution, this will be enforced anyway.

If we drop phi conditions, we do not need to care about loop-carried dependencies as our execution conditions are not related at all and the data flow conditions relate at most two nodes, which never leads to a transitive relation by assumption 3.

4.5 Variable-Values Satisfying the Path-Condition

Our modified path condition now states:

Let $CH(\alpha, \beta) = \{p \mid p \text{ is an acyclic path in the program dependence graph from } \alpha \text{ to } \beta\}$. And $D(p)$ as in section 2.5.2.

$$PC'(\alpha, \beta) \equiv \bigvee_{p \in CH(\alpha,\beta)} \Bigg(\underbrace{\Big(\bigwedge_{n \in_V p} \hat{E}(n) \Big)}_{\text{execution condition}} \wedge \underbrace{\Big(\bigwedge_{d \in D(p)} \hat{\delta}(d) \Big)}_{\text{data flow condition}} \Bigg)$$

For later use we will also define:

- $\mathcal{N} = \{n \in N \mid \exists p \in CH(\alpha, \beta) : n \in_V p\}$ the set of relevant nodes for $PC'(\alpha, \beta)$
- For $n \in \mathcal{N} : V(n) = \{v \in \text{Var} \mid \text{such that } \text{tag}_{\text{idx}(n)}(v) \text{ occurs in } PC'(\alpha, \beta)\}$ the set of relevant variables for n.

Now we can describe a property which holds iff variable values during the program execution fulfill the new path condition. We will call this property existence of a fulfilling execution (EFE).

Definition 9. *Existence of a fulfilling execution (EFE) is true if and only if it exists an execution $\xi = (t, i) : \exists N \in\subseteq \mathcal{N} : \forall n \in N : \exists t^n \in prefix(t)$ such that:*

- *t^n ends at n*
- *$\theta(PC'(\alpha, \beta)) \equiv true$*

Where $\theta = \mathop{\diamond}_{n \in N} (val_{t^n,i} \diamond tag^{-1}_{idx(n)})$ and $tag^{-1}_k(T)$ replaces in T each variable superscripted with k with the variable not superscripted.

EFE means there is a set of nodes and the values of variables at an execution of these nodes satisfy PC'. The subset of $\mathcal{N}$ is needed, as some nodes constrained in PC' may not have been executed in execution ξ (For example see A.4).

If a program has a flow from α to β, then we know from [13] this flow can only occur along a path in the program dependence graph and then all nodes along this path need to be executed and the data flow conditions need to hold at these nodes. In conclusion, with theorem 1 this means:

In a program exists a flow from α to $\beta \Rightarrow$ EFE holds for $PC'(\alpha, \beta)$.

4.6 Example

We will again look at figure 2.1 and analyse the same path as in section 2.5.4: $p = [i(\text{high})] \rightarrow_d^* [19/50] = (i(\text{high}), 7/11, 8/15, 14/37, 16/52, 12/54, 19/50)$. The execution conditions are still true for all nodes except 14/37 and 16/52, but this time we will not take the execution condition from 16/52 into account, as it is a phi node. $\hat{E}(14/37) \equiv i_{56}^{37} < 3 \wedge sum_{54}^{37} = 0$. The only proper sub path for a data dependency condition is still the same $d = (7/11, 8/15, 14/37)$ but $\hat{\delta}(d) \equiv 1 \neq 2 \wedge 1 = i_{56}^{37} \equiv 1 = i_{56}^{37}$.

$$\begin{aligned} PC'(i(\text{high}), 19/50) \equiv& \hat{E}(14/37) \wedge \hat{\delta}(d) \\ \equiv& i_{56}^{37} < 3 \wedge sum_{54}^{37} = 0 \wedge 1 = i_{56}^{37} \end{aligned}$$

This can be read as: There has to be one point during execution, such that $i_{56} < 3 \wedge sum_{54} = 0 \wedge 1 = i_{56}$. Such a point does exist indeed. Let $i = \{\text{low1}/0, \text{high}/1, \text{low2}/2\}$ and let t be the trace such that $\xi = (t, i)$ is an execution. This execution passes 14/37 once in the second iteration. Let p be the prefix of t ending at 14/37.

$$\begin{aligned} & \text{val}_{p,i}(\text{tag}^{-1}_{\text{idx}(14/37)}(PC'(i(\text{high}), 19/50))) \\ \equiv& \text{val}_{p,i}(i_{56} < 3 \wedge sum_{54} = 0 \wedge 1 = i_{56}) \\ \equiv& 1 < 3 \wedge 0 = 0 \wedge 1 = 1 \\ \equiv& \text{true} \end{aligned}$$

We found a fulfilling execution and EFE is true, which is not astonishing as there is actually a flow and we also got this result from the standard path condition. But if all input parameters are larger than zero as in the Sum experiment (A.1), there is no flow and as EFE is not true as well, it will be detected. The standard path condition was not able to detect this.

5. CEGAR for EFE

Counterexample-guided abstraction (CEGAR) refinement was introduced in [2] to handle the state explosion problem in symbolic model checking. In difference to [2] we do not have a formula in linear temporal logic (LTL), but a formula over QF_AUFBV (quantifier-free formulas over the theory of bitvectors, bitvector arrays, and uninterpreted functions), which can be solved using an SMT solver.

We can use CEGAR to provide a sound method to check if EFE holds for a program and two nodes α, β or not. We will first compute the modified path condition $PC'(\alpha, \beta)$. It serves as property for our code model. Our model of the code needs only to contain, which values of variables can occur together at the execution of nodes in $\mathcal{N}$. This could be expressed in an SMT formula M. We would have $\mathrm{cl}_\exists(PC'(\alpha, \beta) \wedge M) \Leftrightarrow$ EFE. Of course to compute M is very inefficient. Instead, we will use an abstraction A of this model, starting with the formula true, which means any values can occur together. CEGAR is now used to refine this abstraction (see figure 5.1).

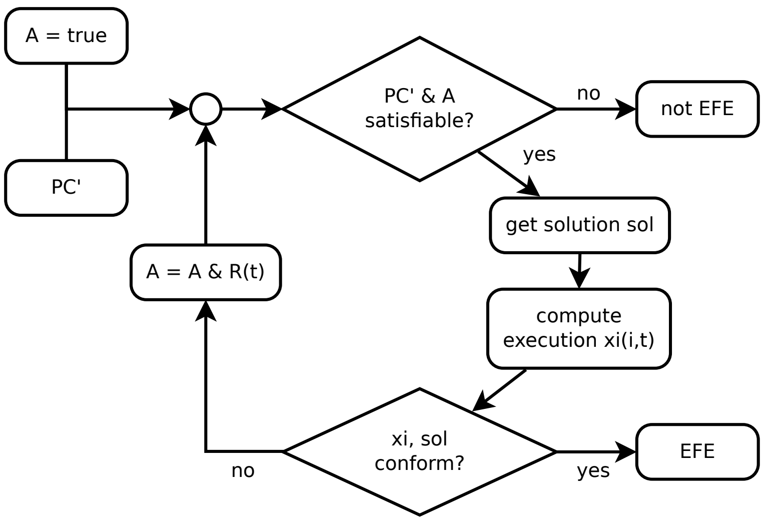


Figure 5.1: CEGAR overview

5.1 Validation

We want a solution to be valid, iff the values of the solution occur during an execution of the program with the input variables from the solution. This means the solution conforms to an execution. Before we can state a formal definition of a valid execution, we need to care about the location where the values of the solution and the execution need to be equal. We must not only look if the value of a variable occurs during the program run, or we might not detect coeval conditions (section 4.2). We do not know if a node $n \in \mathcal{N}$ is executed as not all nodes in $\mathcal{N}$ are executed in an execution that shows EFE. So checking the values of variables at the execution of nodes in $\mathcal{N}$ is not an option either.

For any $n \in \mathcal{N}$ and any trace t we can define $P(t, n) = \{p \in \text{prefix}(t) \mid \exists x \in \text{tail}(n)\colon \text{p ends at } x\}$ the set of all sub paths that end at the tails of loops that contain n. This set is interesting as there is one path for every loop that could have contained n. And variables, which influence the execution of n do not change between their use and the end of the most inner loop:

Theorem 2. *Given a CFG $G_{CFG} = (V, E)$ let $n \in V$ and t be a path in G_{CFG}, which contains n and let Var_n be a set of variables such that their assigning nodes dominate n.*

Let $q \in$ prefix(t), such that q ends at n. Let $p \in P(t, n)$ be the shortest path such that $q \in$ prefix(p). It holds:

$$\forall v \in Var_n : val(q, v) = val(p, v)$$

Proof. Assume there is a $v \in \text{Var}_n$ such that $val(q, v) \neq val(p, v)$. Let s be the unique node assigning v. Then p is of the form:

$$p = \overbrace{(\text{start} \to^* s) \circ \underbrace{(s \to^* n)}_{=:\hat{p}}}^{=q} \circ \overbrace{\underbrace{(n \to^* s)}_{=:\hat{q}} \circ (s \to^* x)}^{=:q'} \qquad \underbrace{\hat{p} \circ \hat{q}}_{=:p'}$$

where $|p'|_s = 2$. There has to be a back edge $(t, h) \in_E p'$ such that $s \in \text{NatLoop}(t, h)$.

Assume $(t, h) \in_E \hat{p}$ so $\hat{p} = (s \to^* t) \circ (t \to h) \circ (h \to^* n)$. As $|\hat{p}|_s = 1$ and h dominates s by lemma 2 there is a path $p'' = \text{start} \to^* h \circ h \to^* n$ with $s \notin_E p''$, which is a contradiction as s dominates n.

Assume $(t, h) \in_E \hat{q}$ so $\hat{q} = (n \to^* t) \circ (t \to h) \circ (h \to^* s)$. It holds h dominates s dominates n, therefore we can construct a path from start over h to n, which does not contain a back edge. We can append $\hat{q}$ to this path, so we reach h again and have a loop. In this case $\hat{q}$ has to contain a back edge and n is in the natural loop of this back edge. This is a contradiction to p being the shortest path in $P(t, n)$ that contains q. $\square$

Using this property we can check the values of the solution at the end of the loops:

Definition 10. *Given a substitution sol, replacing each tagged variable such that $sol(A \wedge PC'(\alpha, \beta)) \equiv true$, sol represents a satisfying solution. Given the execution $\xi(t, i)$ such that for each input parameter v is $i(v) = sol(v)$. ξ is valid with respect to sol iff*

$$\forall n \in \mathcal{N} \exists p \in P(t, n) : \bigwedge_{v \in V(n): val(p,v,i) \neq \bot} : val(p, v, i) = sol(tag_{idx(n)}(v))$$

Notice, that we do not need a solver to check validity but can check it programmatically (see figure 5.2).

This definition can reject executions which have a flow or are showing EFE. That is not a problem as the refinement will ensure that valid executions are not permitted, so we will get this execution again with conforming values. So an invalid execution only states that our abstraction of the code is not good enough. But for a valid execution, we want EFE to hold:

```
procedure isValid($\xi = (t, i)$, sol)
    $M \leftarrow \mathcal{N}$
    for $n \in M$ do
        for $p \in P(t, n)$ do
            allMatched $\leftarrow$ true

            for $v \in V(n)$ : val$(p, v, i) \neq \perp$ do
                if val$(p, v, i) \neq$ sol(tag$_{\mathrm{idx}(n)}(v)$) then
                    allMatched $\leftarrow$ false
                end if
            end for

            if allMatched then
                $M \leftarrow M \setminus \{n\}$
            end if

        end for
    end for
    return $(M = \emptyset)$
end procedure
```

Figure 5.2: Checking Validity

Theorem 3. *If $\xi = (t, i)$ is valid then EFE holds.*

Proof. For $n \in \mathcal{N}$ let $p^n \in P(t, n)$ be the path which showed validity for n in the definition of a valid execution (definition 10). Let $N \subset \mathcal{N}$ be such that $\forall n \in N : val_{p^n,i}(\hat{E}(n)) \equiv true$.

As the execution condition of n is true at the end of the loop n was executed in this loop iteration. So we have $\forall n \in N : n \in_V p^n$. For any $n \in N$ let t^n be the longest prefix of p^n that ends at n. It is $\forall v \in V(n) : val(t^n, v) = val(p^n, v)$ by theorem 2.

We can now construct θ as in the definition of EFE (definition 9). It is $\forall n \in N : \forall v \in V(n) : val(t^n, v) = val(p^n, v) = sol(\mathrm{tag}_{\mathrm{idx}(n)}(v))$. And therefore $\forall n \in N : \forall v \in V(n) : \theta(\mathrm{tag}_{\mathrm{idx}(n)}(v)) = \mathrm{sol}(\mathrm{tag}_{\mathrm{idx}(n)}(v))$. As $\mathrm{sol}(PC'(\alpha, \beta)) \equiv$ true it is also $\theta(PC'(\alpha, \beta)) \equiv$ true. This is because the variables of nodes not in N cannot be part of the solution as $\forall n \in \mathcal{N} \setminus N : sol(E(n)) \equiv false$, so these variables are absorbed anyway. □

5.2 Refinement

During the refinement we want to collect and add the information why the previous solution was not a valid one, to prevent this solution from occuring again.

5.2.1 Refining Single Variables

The basic idea is as in [15]: We will collect the values of a variable that occur and constrain them, such that the refinements of different executions do not contradict each other:

Given a node s which assigns the variable v. Given an execution $\xi = (p, i)$ let $\hat{P}$ be the set of all prefixes of p ending at s.

$$\begin{aligned}
&\bigvee_{\hat{p}\in\hat{P}} v = \text{if guard}(\hat{p}) \text{ then val}(\hat{p}, v) \text{ else ANY} \\
\Leftrightarrow\ &\bigvee_{\hat{p}\in\hat{P}} \text{guard}(\hat{p}) \Rightarrow (v = \text{val}(\hat{p}, v)) \\
\Leftrightarrow\ &\bigvee_{\hat{p}\in\hat{P}} \neg\text{guard}(\hat{p}) \vee (v = \text{val}(\hat{p}, v)) \\
\Leftrightarrow\ &\left(\bigvee_{\hat{p}\in\hat{P}} \neg\text{guard}(\hat{p})\right) \vee \left(\bigvee_{\hat{p}\in\hat{P}} (v = \text{val}(\hat{p}, v))\right) \\
\Leftrightarrow\ &\neg\text{guard}(\max(\hat{P})) \vee \left(\bigvee_{\hat{p}\in\hat{P}} (v = \text{val}(\hat{p}, v))\right)
\end{aligned}$$

The last transformation is correct as the guard of a longer path only conjuncts additional restraints and therefore the conditions from the shorter paths can be absorbed: $\neg a \vee \neg(a \wedge b) \Leftrightarrow \neg(a \wedge b)$.

Given the program from figure 5.3 and an execution $\xi = (p, i)$ with $i = \{a/1\}$ and $p = 2 \to 3 \to 4 \to 3 \to 6$ we would get $0 < a \Rightarrow i_2 = 1$. But if we are executing the program with $i = \{a/2\}$ we will also get the value $i_2 = 2$. The values $a = i_2 = 2$ are not satisfying the generated formula. So the refinement of the first execution would contradict the second execution. To prevent this, we need to add a guard to prevent further loop iterations that is we add $\neg C$ disjunctive where C contains all constraints after the last execution of s:

$$\begin{aligned}
&\neg\text{guard}(\max(\hat{P})) \vee \left(\bigvee_{\hat{p}\in\hat{P}} (v = \text{val}(\hat{p}, v))\right) \vee \neg C \\
\Leftrightarrow\ &\neg\text{guard}(p) \vee \left(\bigvee_{\hat{p}\in\hat{P}} (v = \text{val}(\hat{p}, v))\right) \\
\Leftrightarrow\ &\text{guard}(p) \Rightarrow \left(\bigvee_{\hat{p}\in\hat{P}} (v = \text{val}(\hat{p}, v))\right)
\end{aligned}$$

So what the formula is basically saying is if we have exactly this execution path, then we will get the following symbolic values for v. This seems straight forward but can cause the process to explore all paths through the program as we will see in section 6.

```
1 int foo(int a) {
2         $i_0$ = 0;
3         while ($i_1$ = $\Phi(i_0, i_2)$; $i_1$ < a) {
4                 $i_2 = i_1 + 1$;
5         }
6         return $i_1$;
7 }
```

Figure 5.3: simple refine

5.2.2 Refining Variable Sets

The above formula only tracks single variables, but as we saw in section 4.2 we need to collect information about which values occur together as well. For a specific execution point given by a trace p and a node $n \in \mathcal{N}$

$$\varphi(n,p) := \bigwedge_{v \in V(n): val(p,v) \neq \perp} \mathrm{tag}(v, \mathrm{idx}(n)) = \mathrm{val}(p,v)$$

is a formula describing the symbolic values at p of relevant variables for n.

As in section 5.1 we will collect values at the end of each loop iteration, which could have contained a specific node: For $n \in \mathcal{N}$ let $P(t,n) = \{p \in \mathrm{prefix}(t) \mid \exists x \in \mathrm{tail}(n)$: p ends at $x\}$

$$R(t) := \bigwedge_{n \in \mathcal{N}} \mathrm{guard}(t) \Rightarrow \bigvee_{p \in P(t,n)} \varphi(n,p)$$

We need to ensure that an invalid solution is never found again:

Theorem 4. *Let sol be a solution and let $\xi = (t,i)$ be the corresponding execution as in definition 10. It is: sol$(R(t)) \equiv$ false*

Proof. As the execution is not valid with respect to sol we know:

$$\neg \forall n \in \mathcal{N} \exists p \in P(t,n) : \bigwedge_{v \in V(n): \mathrm{val}(p,v,i) \neq \perp} : \mathrm{val}(p,v,i) = \mathrm{sol}(\mathrm{tag}_{\mathrm{idx}(n)}(v)) \tag{5.1}$$

$$\Leftrightarrow \exists n \in \mathcal{N} : \forall p \in P(t,n) : \neg \bigwedge_{v \in V(n): \mathrm{val}(p,v,i) \neq \perp} : \mathrm{val}(p,v,i) = \mathrm{sol}(\mathrm{tag}_{\mathrm{idx}(n)}(v)) \tag{5.2}$$

$$\Leftrightarrow \exists n \in \mathcal{N} : \forall p \in P(t,n) : \exists v \in V(n) : \mathrm{val}(p,v,i) \neq \perp \wedge \mathrm{val}(p,v,i) \neq \mathrm{sol}(\mathrm{tag}_{\mathrm{idx}(n)}(v)) \tag{5.3}$$

Let n be this one. We only need to show:

$$\mathrm{sol}\left(\mathrm{guard}(t) \Rightarrow \bigvee_{p \in P(t,n)} \varphi(n,p)\right) \equiv \mathrm{false}$$
$$\Leftrightarrow \mathrm{sol}(\mathrm{guard}(t)) \equiv \mathrm{true} \wedge \mathrm{sol}\left(\bigvee_{p \in P(t,n)} \varphi(n,p)\right) \equiv \mathrm{false}$$
$$\Leftrightarrow \mathrm{sol}(\mathrm{guard}(t)) \equiv \mathrm{true} \wedge \forall p \in P(t,n) : \mathrm{sol}(\varphi(n,p)) \equiv \mathrm{false}$$

sol(guard(t)) $\equiv$ true holds as the input of ξ is based on sol and by the properties of ξ being an execution. Given any $p \in P(t,n)$.

$$\mathrm{sol}(\varphi(n,p)) \tag{5.4}$$
$$\equiv \mathrm{sol}(\bigwedge_{v \in V(n): val(p,v) \neq \perp} \mathrm{tag}(v, \mathrm{idx}(n)) = \mathrm{val}(p,v)) \tag{5.5}$$
$$\equiv \bigwedge_{v \in V(n): val(p,v) \neq \perp} \mathrm{sol}(\mathrm{tag}(v, \mathrm{idx}(n))) = \mathrm{sol}(\mathrm{val}(p,v)) \tag{5.6}$$
$$\equiv \bigwedge_{v \in V(n): val(p,v) \neq \perp} \mathrm{sol}(\mathrm{tag}(v, \mathrm{idx}(n))) = \mathrm{val}(p,v,i) \tag{5.7}$$
$$\equiv \mathrm{false} \tag{5.8}$$

(5.6) $\equiv$ (5.7) as val(p,v) does only contain input variables and the substitution sol and i are identical for these. (5.7) $\equiv$ (5.8) as we know from (5.3) there is a v such that the equality does not hold and therefore the whole formula is false. □

We also need to show that an execution which shows EFE provides a solution for the refinement. This solution is provided by the values and execution points that also satisfied the PC.

Theorem 5. *Given a program for which EFE holds. Let $\xi' = (t', i')$, θ as in the definition of EFE (definition 9) and $\xi = (t, i)$ the refined execution. It holds: $\theta(R(t)) \equiv$ true.*

Proof.

$$\text{true} \equiv \theta(R(t)) \equiv \bigwedge_{n \in \mathcal{N}} \theta(\text{guard}(t)) \Rightarrow \bigvee_{p \in P(t,n)} \theta(\varphi(n,p)) \tag{5.9}$$

$$\Leftrightarrow \forall n \in \mathcal{N} : \text{true} \equiv \theta(\text{guard}(t)) \Rightarrow \bigvee_{p \in P(t,n)} \theta(\varphi(n,p)) \tag{5.10}$$

Let n any $n \in \mathcal{N}$. **Case 1**: $\theta(guard(t)) \equiv$ false then the formula is true.

Case 2: $\theta(guard(t)) \equiv$ true $\Leftrightarrow i'(guard(t)) \equiv$ true, as guard(t) only contains parameters as free variables. So we know $t = t'$ by lemma 1.

We have to show $\exists p \in P(t,n) : \theta(\varphi(n,p)) \equiv$ true. We will use t^n, the sub path of t' ending at n, which is used in the definition of θ. Let p be as in theorem 2 for t^n, the path ending at the tail of the loop, which contained n. We know $p \in P(t', n) = P(t, n)$.

$\theta(\varphi(n,p)) \equiv$ true $\Leftrightarrow \forall v \in V(n) : \text{val}(p,v) \neq \perp$: $\theta(\text{tag}_{\text{idx}(n)}(v) = \text{val}(p,v))$ Let v be any $v \in V(n) : \text{val}(p,v) \neq \perp$.

$$\theta(\text{tag}_{\text{idx}(n)}(v) = \text{val}(p,v)) \tag{5.11}$$

$$\equiv \theta(\text{tag}_{\text{idx}(n)}(v)) = \theta(\text{val}(p,v)) \tag{5.12}$$

$$\equiv \theta(\text{tag}_{\text{idx}(n)}(v)) = \text{val}(p,v,i') \tag{5.13}$$

$$\equiv \text{val}_{t^n,i'}(\text{tag}^{-1}_{\text{idx}(n)}(\text{tag}_{\text{idx}(n)}(v))) = \text{val}(p,v,i') \tag{5.14}$$

$$\equiv \text{val}_{t^n,i'}(v) = \text{val}(p,v,i') \tag{5.15}$$

$$\equiv \text{val}(t^n,v,i') = \text{val}(p,v,i') \tag{5.16}$$

(5.13) $\equiv$ (5.14) as $\text{tag}_{\text{idx}(n)}(v)$ does only contain variables tagged with n. (5.16) is true by theorem 2. ☐

As there is only a finite number of solutions and an invalid solution does not show up again, we know the process is terminating. As solutions that show EFE are not permitted, the CEGAR loop will produce such a solution, if one exists.

6. Experiments

The examples are unified to analyse if a flow can occur from the high input parameter to the return statement. Of course we could analyse the flow between two arbitrary nodes.

In figure 6.1 is a table with the experiments and some additional information. For the formula length each variable and each operation counts.

6.1 Sum

This example was introduced by [15] and shows that refinement can be used to increase the precision of path conditions. The input parameters for this example are constrained by $low1 > 0 \wedge high > 0 \wedge low2 > 0$. Therefore line 14 will only be executed once in the first iteration of the loop, where the data flow condition will not hold.

6.2 Coeval

This example shows that it is not enough that the variable values occur, but they need to occur together. (See section 4.2)

6.3 NonCoeval

This example shows how variables are distinguished over different loop iterations. (See section 4.2)

name	exists flow	false positive	itterations	time in s	formula length
Sum	no	no	2	<1	63
Coeval	no	no	2	<1	31
NonCoeval	yes	/	2	<1	43
TwoFlows	yes	/	1	<1	23
ExpRun	no	no	9	<1	221
LoopRun	no	no	$\approx 2^{31}$	>60 years	
ExecutionOrder	no	yes	2	<1	67
Min	no	yes	129	160	125023

Figure 6.1: Overview of Experiments

6.4 TwoFlows

TwoFlows contains two paths for a flow, which exclude each other. Additionally variables from one path do not occur if the other path is taken.

6.5 ExpRun

In section 5.2.1 we noticed that including the whole guard can lead to brute forcing all possible paths, which happens in this example. The number of possible paths is exponential in the number of branches.

6.6 LoopRun

If the number of iterations in a loop depends on an input parameter, the approach becomes absolutely impracticable. In LoopRun this is the case. For an execution path p, guard(p) contains $\bigwedge_{l \in 0..k-1} l < \text{low} \wedge k \geq \text{low}$ where k is the greatest number i takes. As $\neg$guard(p) is part of the refinement a new solution can be found by increasing low by 1. If MaxInt is the largest integer, then LoopRun would need about MaxInt refinement iterations. For Java MaxInt $= 2^{31} - 1 = 2147483647$ this would take way too long.

6.7 ExecutionOrder

We can have still false positives, as this example from [9] shows. There is a possible flow along the nodes $(i(\text{high}), 10/18, 7/37, 7/41, 12/23, 7/35, 7/39, 14/33)$. All of these nodes are executed and no data flow conditions exist, so the refinement reaches a valid state. However the node (10/18) is only executed after (12/23) but would need to be executed before for a flow.

6.8 Min

A basic method returning the minimum of all input values. There is no flow with an additional assertion: High is the largest value. But our approach provides a false positive, so lets have a look what is happening.

$\text{result}^{104}_{108} = 4$ is the PC for the second path found. As $\text{result}^{104}_{108} = 8 \neq 4$ the solution must satisfy the PC of the first path. The first path basically consists of the array assignment (12/35) the if-statement (21/82), the assignment of result (22/87) and the return statement (27/104). If this path contains a flow, it does not necessarily mean that the high value is returned, but that result is influenced by high.

The path condition for the first path contains the execution condition for the if statement $E(21/82)$ and its data flow condition (all conditions containing variables superscripted with 82) as well as the execution condition for the assignment of result $E(22/87)$ (all conditions containing variables superscripted with 87). The solution provides $i^{82}_{110} = 4$, so the variables satisfying the conditions for the if-statement are from the fourth iteration of the loop, where of course the statement is executed and a comparison to the array field containing the high variable is done. However, this comparison will always be wrong as $high = x[4] \geq x[result]$, no matter which value result has. This also means in this iteration the flow from (21/82) to (22/87) does never happen.

Why did we still get this false positive solution? (22/87) is executed, but in the first iteration ($i^{87}_{110} = 1$). Our construction of the PC did not contain the information that the if-statement and the assignment of result need to happen in the same iteration. This provides another idea how to increase the precision of the PC: A flow along a control dependency edge indicates that both nodes need to be executed in the same iteration of a loop.

But for the moment we can also add our analysis to the constraint: We know the flow can not happen in the fourth iteration so we can safely assert $i^{82}_{110} \neq 4$. Given this additional assertion the CEGAR-loop is terminated with no more satisfying solutions. We know now that there is no flow.

7. Conclusion

We provided a detailed method to ensure that we generate witnesses that fulfill the path condition and are representing a concrete execution. As data shows this approach lags scalability as the number of iterations corresponds to the number of possible execution paths. This is due to using the full guard in the refinement. In future work this problem should be tackled for example by reducing the guard to the conditions of statements that can influence the analysed variables. Also we do not need to track all variables, if a subset of variables of a node is enough to prevent the invalid solution to occur again.

$$\bigwedge_{n \in \mathcal{N}} \bigwedge_{V' \in \mathcal{P}(V(n))} \text{guard}_{|V'}(t) \Rightarrow \bigvee_{p \in \mathcal{P}(t,n)} \bigwedge_{v \in V'} v = \text{val}(p, v)$$

This would reduce the number of needed iterations. To reduce the time needed by the solver to produce a satisfying solution, incremental solving should be used instead of always passing the whole formula. Also unsatcore and automatic procedures to reduce the length of the refinement can help to reduce the solvers runtime.

It is also possible to add more timing information in a similar way as for coeval conditions. For example a higher superscript could indicate that values need to occur after other values, earlier occurrences would be ignored in the refinement. This would increase precision further.

The original PC as in [12, 13, 14] is less precise but more scalable, as it did not contain the refinement loop. The approach on temporal path conditions from [9] is even more precise, but lags scalability as well and some cases considered in the LTL approach can be added to this approach, too.

Compared to other approaches for IFC as type systems, dynamical approaches which monitor the program execution to detect flows or hybrid solutions (an overview can be found in [6]) the described algorithm is not practicable, yet. If the problem of scalability gets solved, then it allows a very fine grained declassification (i.e. allowing flows to happen under certain circumstances) while having view false positives and a witness for each possible flow.

The witnesses produced by the described approach could also be used for other problem domains as test case generation [7] or checking if a node is executable similar to [10].

Bibliography

[1] Corrado Böhm and Giuseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". In: *Commun. ACM* 9.5 (1966), pp. 366–371.

[2] Edmund M. Clarke et al. "Counterexample-Guided Abstraction Refinement". In: *Proceedings of the 12th International Conference on Computer Aided Verification.* 2000, pp. 154–169.

[3] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), pp. 451–490.

[4] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (1987), pp. 319–349.

[5] Joseph A Goguen and José Meseguer. "Security policies and security models". In: *2012 IEEE Symposium on Security and Privacy.* IEEE Computer Society. 1982, pp. 11–11.

[6] Daniel Hedin and Andrei Sabelfeld. "A perspective on information-flow control". In: *Proc. of the 2011 Marktoberdorf Summer School. IOS Press* (2011).

[7] Hyoung Seok Hong et al. "Data flow testing as model checking". In: *Software Engineering, 2003. Proceedings. 25th International Conference on.* IEEE. 2003, pp. 232–242.

[8] Jens Krinke. "Advanced Slicing of Sequential and Concurrent Programs". PhD thesis. Universität Passau, 2003.

[9] Andreas Lochbihler and Gregor Snelting. "On Temporal Path Conditions in Dependence Graphs". In: *Journal of Automated Software Engineering* 16.2 (2009), pp. 263–290.

[10] Kin-Keung Ma et al. "Directed Symbolic Execution." In: *SAS.* Ed. by Eran Yahav. Vol. 6887. 2011, pp. 95–111.

[11] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann, 2007.

[12] Torsten Robschink. "Pfadbedingungen in Abhängigkeitsgraphen und ihre Anwendung in der Softwaresicherheitstechnik". PhD thesis. Universität Passau, 2005.

[13] Gregor Snelting. "Combining Slicing and Constraint Solving for Validation of Measurement Software". In: *Static Analysis.* Sept. 1996, pp. 332–348.

[14] Gregor Snelting, Torsten Robschink, and Jens Krinke. "Efficient Path Conditions in Dependence Graphs for Software Safety Analysis". In: *ACM Transactions on Software Engineering and Methodology* 15.4 (2006), pp. 410–457.

[15] Mana Taghdiri, Gregor Snelting, and Carsten Sinz. “Information Flow Analysis via Path Condition Refinement”. In: *7th International Workshop on Formal Aspects in Security and Trust (FAST)*. 2010, pp. 65–79.

Appendix

A Examples

A.1 Sum

Listing A.1: Java Code of Sum

```java
4   public static int foo(int low1, int high, int low2) {
5       int[] a = new int[3];
6       a[0] = low1;
7       a[1] = high;
8       a[2] = low2;
9
10      int sum = 0;
11      int i = 0;
12      while (i < 3) {
13          if (sum == 0) {
14              sum = sum + a[i];
15          }
16          i++;
17      }
18
19      return sum;
20  }
```

Listing A.2: SSA Code of Sum

```
 3 $a_{3}$ = new int[3];
 7 $a_{7}$[0] = low1;
11 $a_{11}$[1] = high;
15 $a_{15}$[2] = low2;
17 $sum_{17}$ = 0;
20 $i_{20}$ = 0;
22 goto 45;
27 if ($sum_{54}$ $\neq$ 0) goto 39;
37 $sum_{37}$ = $sum_{54}$ + $a_{15}$[$i_{56}$];
39 $sum_{52}$ = $\Phi$($sum_{54}$, $sum_{37}$);
   $i_{39}$ = $i_{56}$ + 1;
45 $i_{56}$ = $\Phi$($i_{20}$, $i_{39}$);
   $sum_{54}$ = $\Phi$($sum_{52}$, $sum_{17}$);
   if ($i_{56}$ < 3) goto 27;
50 return $sum_{54}$;
```

Listing A.3: Program Output of Sum

```
found 1 paths from i(high) to 19 / 50:
[i(high), 7 / 11, 8 / 15, 14 / 37, 16 / 52, 12 / 54, 19 / 50]

path condition with constraints:
($sum_{54}^{37}$ = 0) & ($i_{56}^{37}$ < 3) & (1 = $i_{56}^{37}$) & (low1 > 0) & (high > 0) & (low2 > 0)

 = iteration 1 =
solution: {high = 2147483647, $i_{56}^{37}$ = 1, low1 = 2147483647, low2 = 2147483647, $sum_{54}^{37}$ =
     0}
refinement: (low1 = 0) | (($sum_{54}^{37}$ = 0) & ($i_{56}^{37}$ = 0)) | (($sum_{54}^{37}$ = low1) & ($i_{56}^{37}$ = 1)) |
    (($sum_{54}^{37}$ = low1) & ($i_{56}^{37}$ = 2))
 = iteration 2 =
No more satisfying solutions. No flow.

=======
formula length: 63
iterations: 2
time used: 0.051s
thereof time for SMT solver: 0.035s
```

Figure A.1: Control Flow Graph of Sum

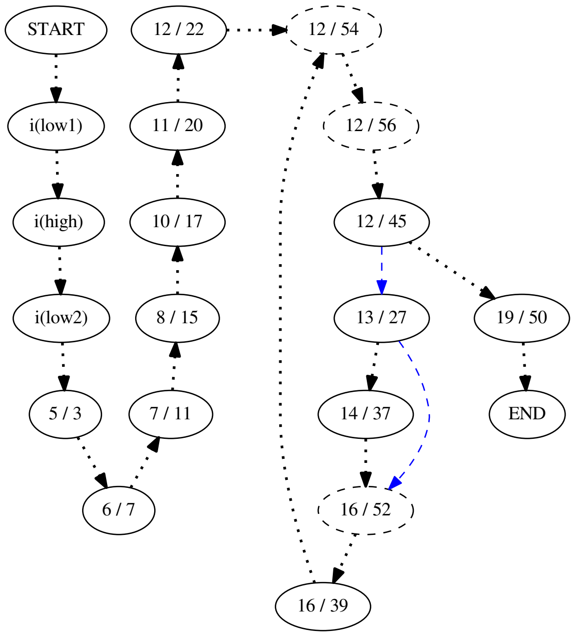
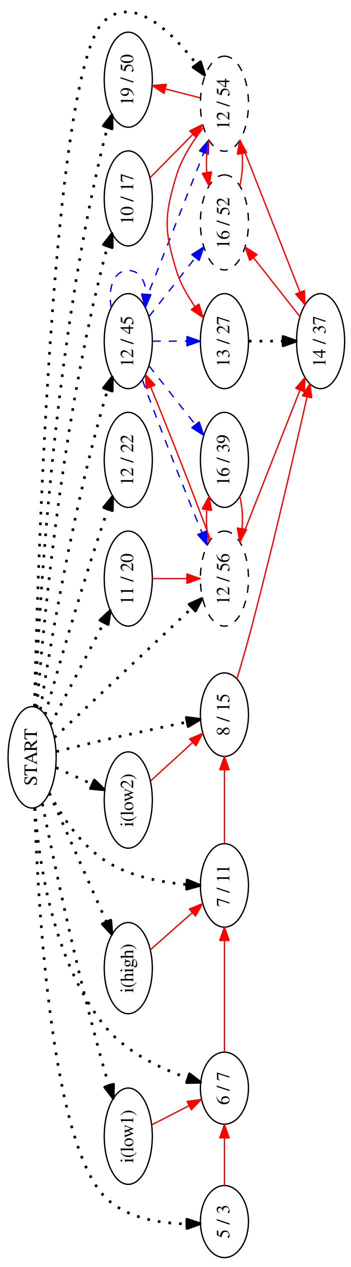


Figure A.2: Program Dependence Graph of Sum



A.2 Coeval

Listing A.4: Java Code of Coeval

```
 4  public static int foo(int high) {
 5      int result = 0;
 6      int e = 0;
 7      int i = 0;
 8      while (i < 2) {
 9          if ((i == 0) && (e == 1))
10              result = high;
11          e++;
12          i++;
13      }
14
15      return result;
16  }
```

Listing A.5: SSA Code of Coeval

```
 1 result$_{1}$ = 0;
 3 e$_{3}$ = 0;
 5 i$_{5}$ = 0;
 6 goto 28;
10 if (i$_{40}$ ≠ 0) goto 20;
15 if (e$_{38}$ ≠ 1) goto 20;
19 result$_{19}$ = high;
20 result$_{34}$ = Φ(result$_{19}$, result$_{36}$);
   e$_{20}$ = e$_{38}$ + 1;
23 i$_{23}$ = i$_{40}$ + 1;
28 i$_{40}$ = Φ(i$_{5}$, i$_{23}$);
   e$_{38}$ = Φ(e$_{20}$, e$_{3}$);
   result$_{36}$ = Φ(result$_{34}$, result$_{1}$);
   if (i$_{40}$ < 2) goto 10;
32 return result$_{36}$;
```

Listing A.6: Program Output of Coeval

```
found 1 paths from i(high) to 15 / 32:
[i(high), 10 / 19, 11 / 34, 8 / 36, 15 / 32]

path condition with constraints:
(e$^{19}_{38}$ = 1) & (i$^{19}_{40}$ = 0) & (i$^{19}_{40}$ < 2)

 = iteration 1 =
solution: {e$^{19}_{38}$ = 1, high = -1, i$^{19}_{40}$ = 0}
refinement: ((e$^{19}_{38}$ = 0) & (i$^{19}_{40}$ = 0)) | ((e$^{19}_{38}$ = 1) & (i$^{19}_{40}$ = 1))
 = iteration 2 =
No more satisfying solutions. No flow.

=======
formula length: 31
iterations: 2
time used: 0.017s
thereof time for SMT solver: 0.011s
```

Figure A.3: Control Flow Graph of Coeval

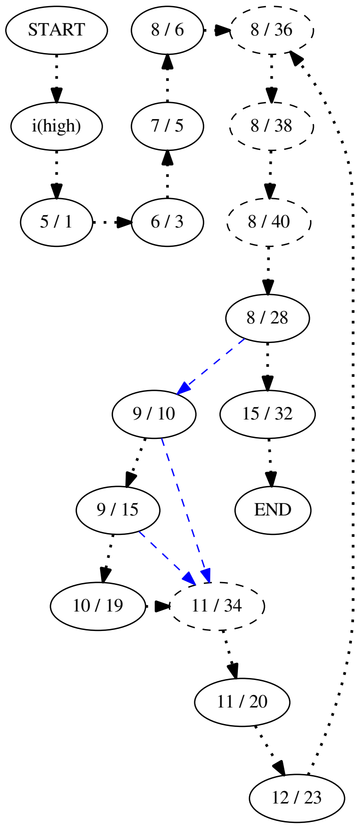
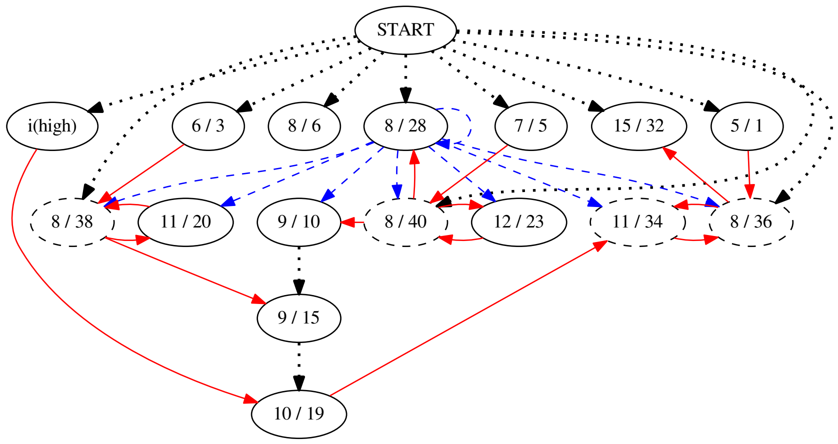


Figure A.4: Program Dependence Graph of Coeval



A.3 NonCoeval

Listing A.7: Java Code of NonCoeval

```
 4  public static int foo(int high) {
 5      int result = 0;
 6      int tmp = 0;
 7      int e = 0;
 8      int i = 0;
 9      while (i < 2) {
10          if (i == 0)
11              tmp = high;
12
13          if (e == 1)
14              result = tmp;
15          e++;
16          i++;
17      }
18
19      return result;
20  }
```

Listing A.8: SSA Code of NonCoeval

```
 1 result$_{1}$ = 0;
 3 tmp$_{3}$ = 0;
 5 e$_{5}$ = 0;
 7 i$_{7}$ = 0;
 9 goto 35;
14 if (i$_{51}$ $\neq$ 0) goto 21;
18 tmp$_{18}$ = high;
21 tmp$_{41}$ = $\Phi$(tmp$_{18}$, tmp$_{47}$);
   if (e$_{49}$ $\neq$ 1) goto 26;
25 result$_{25}$ = tmp$_{41}$;
26 result$_{43}$ = $\Phi$(result$_{25}$, result$_{45}$);
   e$_{26}$ = e$_{49}$ + 1;
29 i$_{29}$ = i$_{51}$ + 1;
35 i$_{51}$ = $\Phi$(i$_{7}$, i$_{29}$);
   e$_{49}$ = $\Phi$(e$_{5}$, e$_{26}$);
   tmp$_{47}$ = $\Phi$(tmp$_{41}$, tmp$_{3}$);
   result$_{45}$ = $\Phi$(result$_{43}$, result$_{1}$);
   if (i$_{51}$ < 2) goto 14;
39 return result$_{45}$;
```

Listing A.9: Program Output of NonCoeval

```
found 1 paths from i(high) to 19 / 39:
[i(high), 11 / 18, 13 / 41, 14 / 25, 15 / 43, 9 / 45, 19 / 39]

path condition with constraints:
(i$^{18}_{51}$ = 0) & (i$^{18}_{51}$ < 2) & (e$^{25}_{49}$ = 1) & (i$^{25}_{51}$ < 2)

 = iteration 1 =
solution: {e$^{25}_{49}$ = 1, high = -1, i$^{18}_{51}$ = 0, i$^{25}_{51}$ = -1}
refinement: ((i$^{18}_{51}$ = 0) | (i$^{18}_{51}$ = 1)) & (((i$^{25}_{51}$ = 0) & (e$^{25}_{49}$ = 0)) | ((i$^{25}_{51}$ = 1) & (e$^{25}_{49}$ =
    1)))
 = iteration 2 =
solution: {e$^{25}_{49}$ = 1, high = -1, i$^{18}_{51}$ = 0, i$^{25}_{51}$ = 1}

Valid solution found!

=======
formula length: 43
iterations: 2
time used: 0.051s
thereof time for SMT solver: 0.011s
```

Figure A.5: Control Flow Graph of NonCoeval

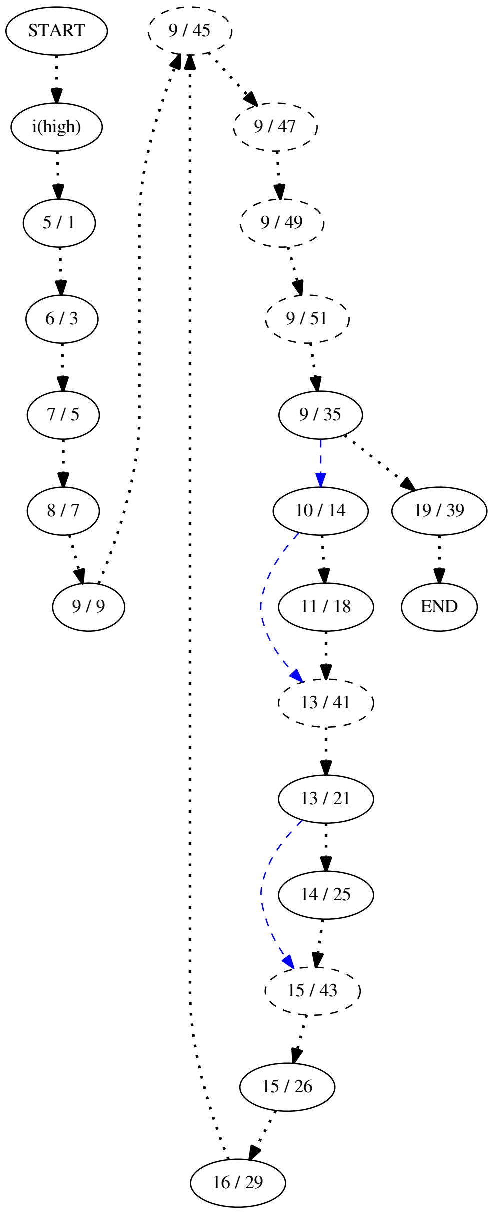
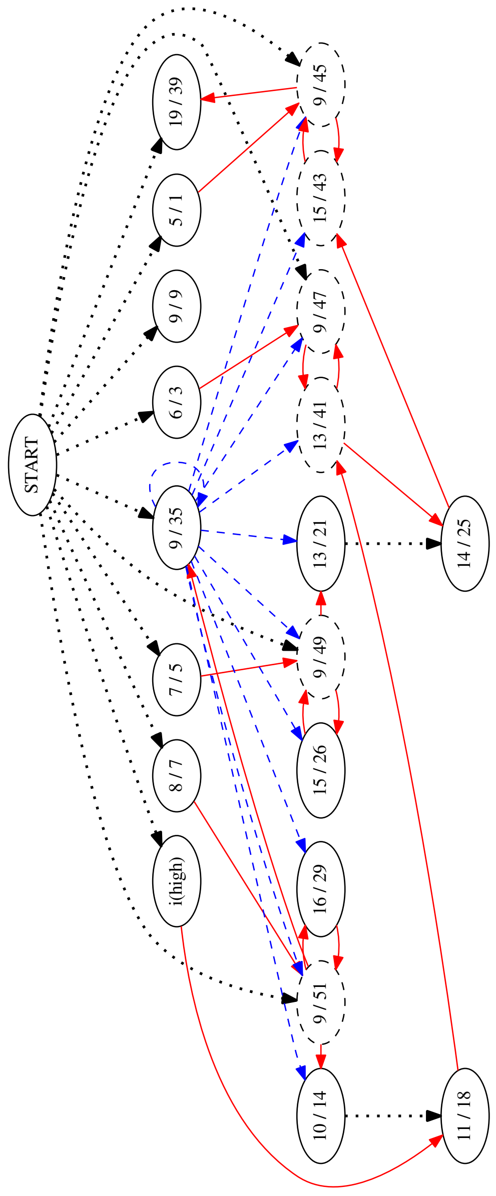


Figure A.6: Program Dependence Graph of NonCoeval



A.4 TwoFlows

Listing A.10: Java Code of TwoFlows

```
 4  public static int foo(int low, int high) {
 5      int result = 0;
 6      int i = 0;
 7      if (low == 0) {
 8          i = 1;
 9          if (i == 1)
10              result = high;
11      } else {
12          i = 2;
13          if (i == 2)
14              result = high;
15      }
16      return result;
17  }
```

Listing A.11: SSA Code of TwoFlows

```
 1 result_1 = 0;
 3 i_3 = 0;
 5 if (low ≠ 0) goto 21;
 9 i_9 = 1;
12 if (i_9 ≠ 1) goto 30;
16 result_16 = high;
17 goto 30;
21 i_21 = 2;
24 if (i_21 ≠ 2) goto 30;
28 result_28 = high;
30 i_34 = Φ(i_21, i_9);
   result_32 = Φ(result_28, result_16, result_1);
   return result_32;
```

Listing A.12: Program Output of TwoFlows

```
found 2 paths from i(high) to 16 / 30:
[i(high), 14 / 28, 16 / 32, 16 / 30]
[i(high), 10 / 16, 16 / 32, 16 / 30]

path condition with constraints:
((i_21^28 = 2) & (low^28 ≠ 0)) | ((i_9^16 = 1) & (low^16 = 0))

 = iteration 1 =
solution: {high = -1, i_21^28 = 2, i_9^16 = -1, low = -1}

Valid solution found!

=======
formula length: 23
iterations: 1
time used: 0.015s
thereof time for SMT solver: 0.013s
```

Figure A.7: Control Flow Graph of TwoFlows

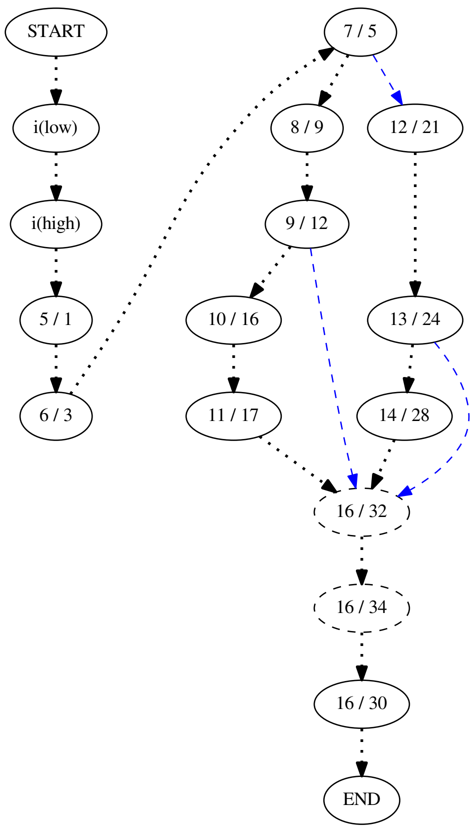
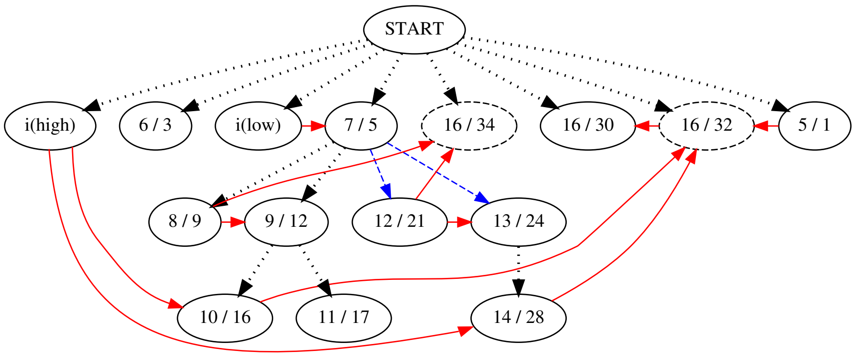


Figure A.8: Program Dependence Graph of TwoFlows



A.5 ExpRun

Listing A.13: Java Code of ExpRun

```
 4  public static int foo(int a, int b, int c, int high) {
 5      int i = 0;
 6      if (a == 0) {
 7          i++;
 8      }
 9      if (b == 0) {
10          i++;
11      }
12      if (c == 0) {
13          i++;
14      }
15
16      int result = a;
17      if (result != a) {
18          result = high;
19      }
20
21      return result;
22  }
```

Listing A.14: SSA Code of ExpRun

```
 1 i$_{1}$ = 0;
 4 if (a $\neq$ 0) goto 11;
 7 i$_{7}$ = i$_{1}$ + 1;
11 i$_{40}$ = $\Phi$(i$_{7}$, i$_{1}$);
   if (b $\neq$ 0) goto 18;
14 i$_{14}$ = i$_{40}$ + 1;
18 i$_{42}$ = $\Phi$(i$_{40}$, i$_{14}$);
   if (c $\neq$ 0) goto 25;
21 i$_{21}$ = i$_{42}$ + 1;
25 i$_{44}$ = $\Phi$(i$_{21}$, i$_{42}$);
   result$_{25}$ = a;
30 if (result$_{25}$ = a) goto 38;
34 result$_{34}$ = high;
38 result$_{46}$ = $\Phi$(result$_{25}$, result$_{34}$);
   return result$_{46}$;
```

Listing A.15: Program Output of ExpRun

```
found 1 paths from i(high) to 21 / 38:
[i(high), 18 / 34, 21 / 46, 21 / 38]

path condition with constraints:
(result$_{25}^{34}$ $\neq$ a$^{34}$)

 = iteration 1 =
solution: {a = -1, b = -1, c = -1, high = -1, result$_{25}^{34}$ = -2}
refinement: (a = 0) | (b = 0) | (c = 0) | (a $\neq$ a) | ((result$_{25}^{34}$ = a) & (a$^{34}$ = a))
 = iteration 2 =
solution: {a = 0, b = -1, c = -1, high = -1, result$_{25}^{34}$ = -1}
refinement: (a $\neq$ 0) | (b = 0) | (c = 0) | (a $\neq$ a) | ((result$_{25}^{34}$ = a) & (a$^{34}$ = a))
 = iteration 3 =
solution: {a = 0, b = 0, c = -1, high = -1, result$_{25}^{34}$ = -1}
refinement: (a $\neq$ 0) | (b $\neq$ 0) | (c = 0) | (a $\neq$ a) | ((result$_{25}^{34}$ = a) & (a$^{34}$ = a))
 = iteration 4 =
solution: {a = 0, b = 0, c = 0, high = -1, result$_{25}^{34}$ = -1}
refinement: (a $\neq$ 0) | (b $\neq$ 0) | (c $\neq$ 0) | (a $\neq$ a) | ((result$_{25}^{34}$ = a) & (a$^{34}$ = a))
 = iteration 5 =
solution: {a = 0, b = -1, c = 0, high = -1, result$_{25}^{34}$ = -1}
refinement: (a $\neq$ 0) | (b = 0) | (c $\neq$ 0) | (a $\neq$ a) | ((result$_{25}^{34}$ = a) & (a$^{34}$ = a))
 = iteration 6 =
solution: {a = -1, b = 0, c = 0, high = -1, result$_{25}^{34}$ = -2}
refinement: (a = 0) | (b $\neq$ 0) | (c $\neq$ 0) | (a $\neq$ a) | ((result$_{25}^{34}$ = a) & (a$^{34}$ = a))
 = iteration 7 =
solution: {a = -1, b = 0, c = -1, high = -1, result$_{25}^{34}$ = -2}
```

```
refinement: (a = 0) | (b ≠ 0) | (c = 0) | (a ≠ a) | ((result$^{34}_{25}$ = a) & (a$^{34}$ = a))
 = iteration 8 =
solution: {a = -1, b = -1, c = 0, high = -1, result$^{34}_{25}$ = -2}
refinement: (a = 0) | (b = 0) | (c ≠ 0) | (a ≠ a) | ((result$^{34}_{25}$ = a) & (a$^{34}$ = a))
 = iteration 9 =
No more satisfying solutions. No flow.

=======
formula length: 211
iterations: 9
time used: 0.15s
thereof time for SMT solver: 0.111s
```

Figure A.9: Control Flow Graph of ExpRun

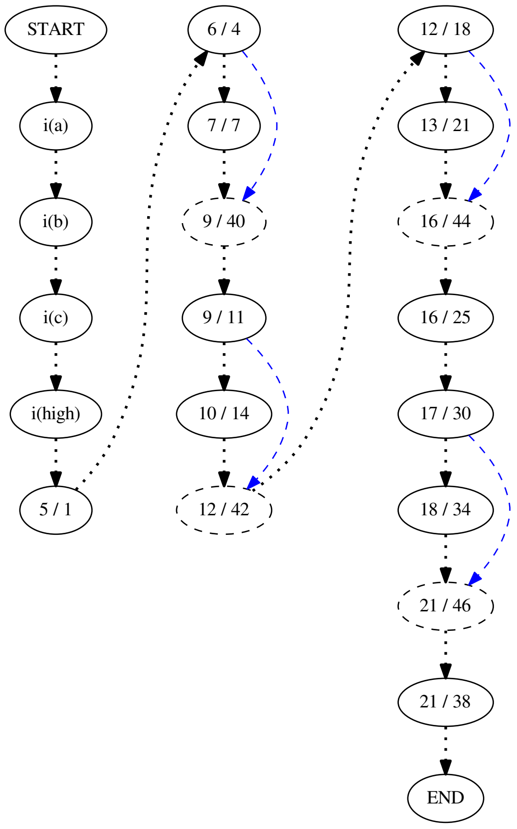
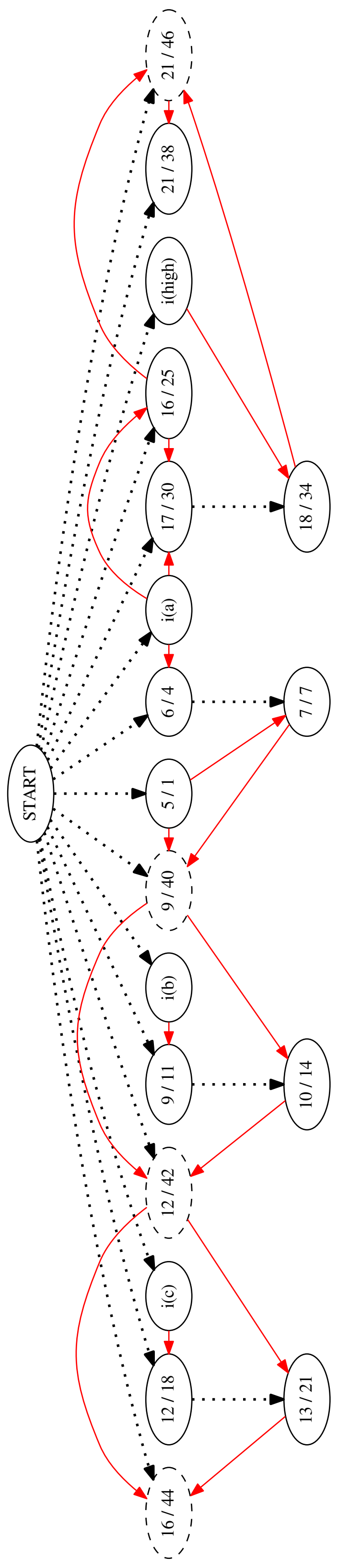


Figure A.10: Program Dependence Graph of ExpRun



A.6 LoopRun

Listing A.16: Java Code of LoopRun

```
 4  public static int foo(int low, int high) {
 5      int i = 0;
 6      int result = 0;
 7
 8      if (low > 0)
 9          while (i < low) {
10              i++;
11          }
12
13      int e = 0;
14      if (e == 1 && i == 1) {
15          result = high;
16      }
17
18      return result;
19  }
```

Listing A.17: SSA Code of LoopRun

```
 1 i_1 = 0;
 3 result_3 = 0;
 5 if (low ≤ 0) goto 20;
 8 goto 16;
11 i_11 = i_38 + 1;
16 i_38 = Φ(i_11, i_1);
   if (i_38 < low) goto 11;
20 i_40 = Φ(i_38, i_1);
   e_20 = 0;
25 if (e_20 ≠ 1) goto 36;
30 if (i_40 ≠ 1) goto 36;
34 result_34 = high;
36 result_42 = Φ(result_3, result_34);
   return result_42;
```

Listing A.18: Program Output of LoopRun

```
found 1 paths from i(high) to 18 / 36:
[i(high), 15 / 34, 18 / 42, 18 / 36]

path condition with constraints:
(i_40^34 = 1) & (e_20^34 = 1)

[Further outputs are not generated, as the second iteration executes the loop with
    MaxInt iterations, which takes too long.]
```

Figure A.11: Control Flow Graph of LoopRun

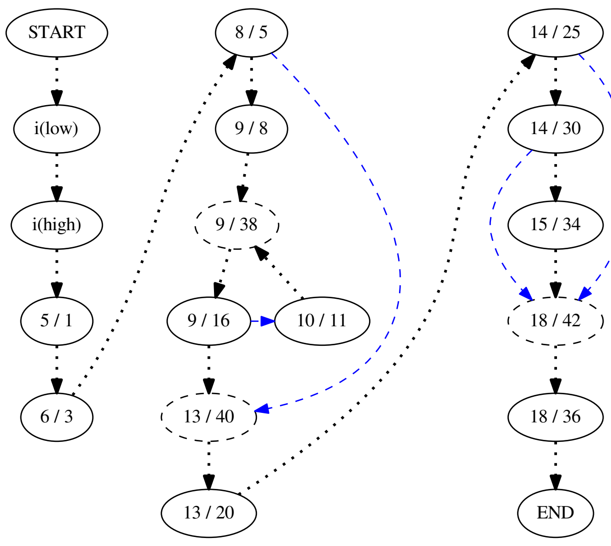
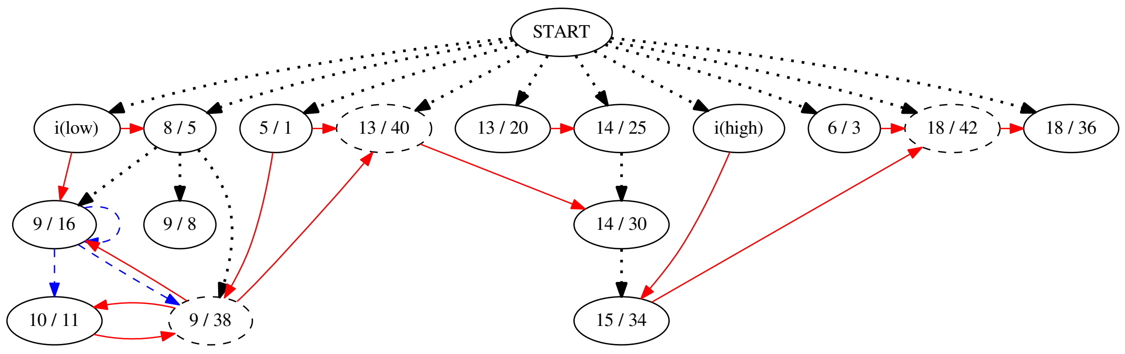


Figure A.12: Program Dependence Graph of LoopRun



A.7 ExecutionOrder

Listing A.19: Java Code of ExecutionOrder

```
 4  public static int foo(int high) {
 5      int y = 0;
 6      int x = 0;
 7      for (int i = 0; i < 5; i++) {
 8          i = i + 1;
 9          if (i > 4)
10              x = high;
11          else
12              y = x;
13      }
14      return y;
15  }
```

Listing A.20: SSA Code of ExecutionOrder

```
 1 y_1 = 0;
 3 x_3 = 0;
 5 i_5 = 0;
 6 goto 29;
 9 i_9 = i_43 + 1;
14 if (i_9 ≤ 4) goto 23;
18 x_18 = high;
19 goto 24;
23 y_23 = x_41;
24 x_37 = Φ(x_41, x_18);
   y_35 = Φ(y_39, y_23);
   i_24 = i_9 + 1;
29 i_43 = Φ(i_5, i_24);
   x_41 = Φ(x_3, x_37);
   y_39 = Φ(y_35, y_1);
   if (i_43 < 5) goto 9;
33 return y_39;
```

Listing A.21: Program Output of ExecutionOrder

```
found 1 paths from i(high) to 14 / 33:
[i(high), 10 / 18, 7 / 37, 7 / 41, 12 / 23, 7 / 35, 7 / 39, 14 / 33]

path condition with constraints:
(i_9^18 > 4) & (i_43^18 < 5) & (i_9^23 ≤ 4) & (i_43^23 < 5)

 = iteration 1 =
solution: {high = -1, i_43^18 = -1, i_43^23 = -1, i_9^18 = 2147483647, i_9^23 = -1}
refinement: (((i_43^18 = 0) & (i_9^18 = 1)) | ((i_43^18 = 2) & (i_9^18 = 3)) | ((i_43^18 = 4) & (i_9^18 =
    5))) & (((i_9^23 = 1) & (i_43^23 = 0)) | ((i_9^23 = 3) & (i_43^23 = 2)) | ((i_9^23 = 5) & (i_43^23 = 4)
    ))
 = iteration 2 =
solution: {high = -1, i_43^18 = 4, i_43^23 = 0, i_9^18 = 5, i_9^23 = 1}

Valid solution found!

=======
formula length: 67
iterations: 2
time used: 0.046s
thereof time for SMT solver: 0.035s
```

Figure A.13: Control Flow Graph of ExecutionOrder

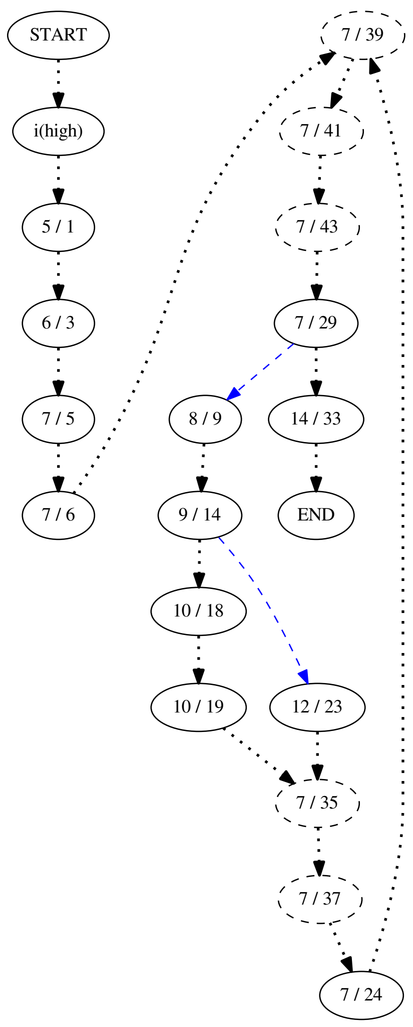
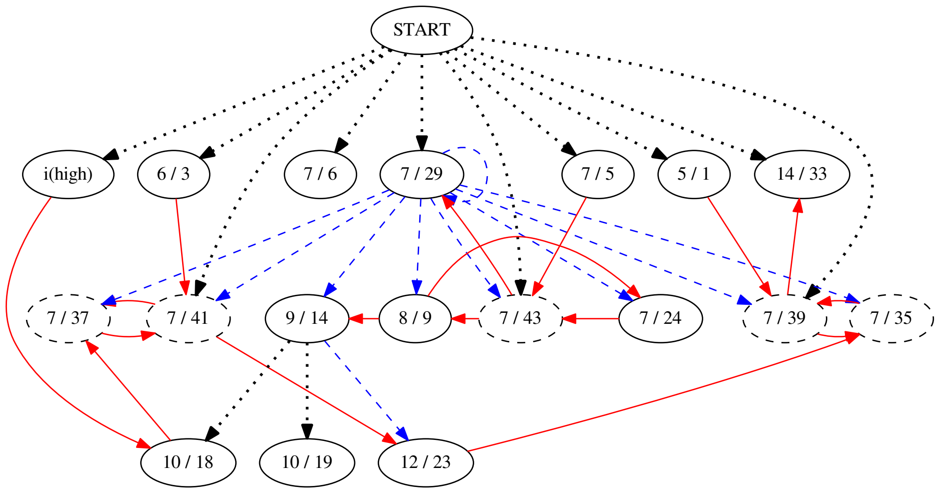


Figure A.14: Program Dependence Graph of ExecutionOrder



A.8 Min

Listing A.22: Java Code of Min

```
 4  public static int foo(int a, int b, int c, int d, int e, int f, int g,
 5          int h, int high) {
 6      int len = 9;
 7      int[] x = new int[len];
 8      x[0] = a;
 9      x[1] = b;
10      x[2] = c;
11      x[3] = d;
12      x[4] = high;
13      x[5] = e;
14      x[6] = f;
15      x[7] = g;
16      x[8] = h;
17
18      int result = 0;
19      int i = 0;
20      while (i < len) {
21          if (x[i] < x[result]) {
22              result = i;
23          }
24          i++;
25      }
26
27      return x[result];
28  }
```

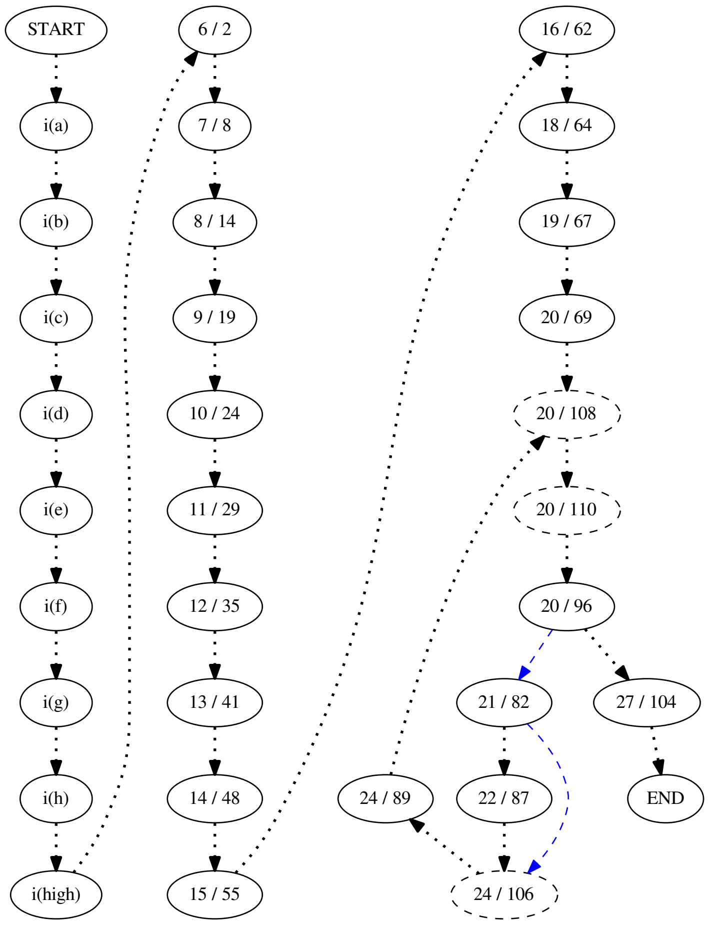
Listing A.23: SSA Code of Min

```
  2 len$_{2}$ = 9;
  8 x$_{8}$ = new int[len_2];
 14 x$_{14}$[0] = a;
 19 x$_{19}$[1] = b;
 24 x$_{24}$[2] = c;
 29 x$_{29}$[3] = d;
 35 x$_{35}$[4] = high;
 41 x$_{41}$[5] = e;
 48 x$_{48}$[6] = f;
 55 x$_{55}$[7] = g;
 62 x$_{62}$[8] = h;
 64 result$_{64}$ = 0;
 67 i$_{67}$ = 0;
 69 goto 96;
 82 if (x$_{62}$[i$_{110}$] $\geq$ x$_{62}$[result$_{108}$]) goto 89;
 87 result$_{87}$ = i$_{110}$;
 89 result$_{106}$ = Φ(result$_{108}$, result$_{87}$);
    i$_{89}$ = i$_{110}$ + 1;
 96 i$_{110}$ = Φ(i$_{67}$, i$_{89}$);
    result$_{108}$ = Φ(result$_{106}$, result$_{64}$);
    if (i$_{110}$ < len$_{2}$) goto 82;
104 return x$_{62}$[result$_{108}$];
```

Listing A.24: Program Output of Min

```
found 2 paths from i(high) to 27 / 104:
[i(high), 12 / 35, 13 / 41, 14 / 48, 15 / 55, 16 / 62, 21 / 82, 22 / 87, 24 / 106,
    20 / 108, 27 / 104]
[i(high), 12 / 35, 13 / 41, 14 / 48, 15 / 55, 16 / 62, 27 / 104]

path condition with constraints:
(((i$^{82}_{110}$ < len$^{82}_{2}$) & (x$^{87}_{62}$[i$^{87}_{110}$] < x$^{87}_{62}$[result$^{87}_{108}$]) & (i$^{87}_{110}$ < len$^{87}_{2}$) & ((4 = i$^{82}_{110}$) | (4 =
    result$^{82}_{108}$))) | (4 = result$^{104}_{108}$)) & (a < high) & (b < high) & (c < high) & (d <
    high) & (e < high) & (f < high) & (g < high) & (h < high)

solution: {a = 1073741823, b = -1, c = -134217729, d = -268435457, e = -536870913,
    f = -1073741825, g = -1342177281, h = -1610612737, high = 2147483647, i$^{82}_{110}$ = 4,
    i$^{87}_{110}$ = 1, len$^{82}_{2}$ = 9, len$^{87}_{2}$ = 9, result$^{104}_{108}$ = 8, result$^{82}_{108}$ = 3, result$^{87}_{108}$ = 0, x$^{87}_{62}$[0]
    = 1073741823, x$^{87}_{62}$[1] = -1, x$^{87}_{62}$[2] = -134217729, x$^{87}_{62}$[3] = -268435457, x$^{87}_{62}$[4] =
    2147483647, x$^{87}_{62}$[5] = -536870913, x$^{87}_{62}$[6] = -1073741825, x$^{87}_{62}$[7] = -1342177281, x$^{87}_{62}$
    [8] = -1610612737}

Valid solution found!

=======
formula length: 125023
iterations: 129
time used: 161.213s
thereof time for SMT solver: 156.192s
```

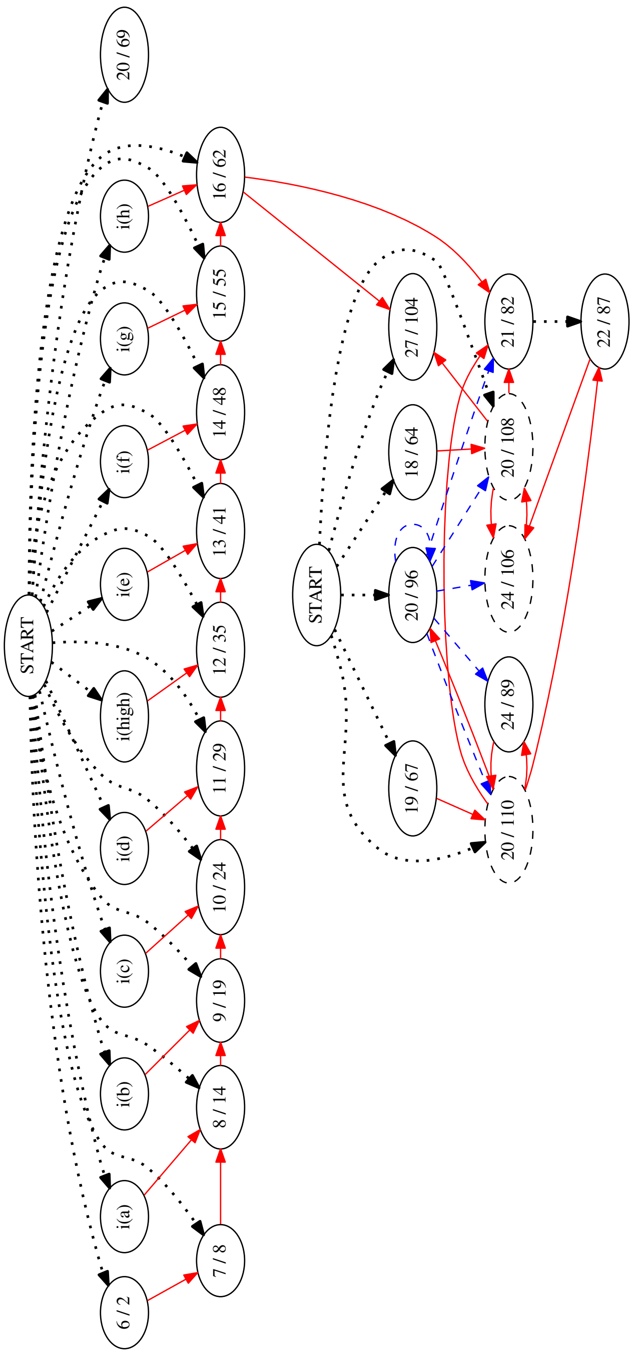
Figure A.15: Control Flow Graph of Min



Listing A.25: Program Output of Min with additional Constraint

```
found 2 paths from i(high) to 27 / 104:
[i(high), 12 / 35, 13 / 41, 14 / 48, 15 / 55, 16 / 62, 21 / 82, 22 / 87, 24 / 106,
    20 / 108, 27 / 104]
[i(high), 12 / 35, 13 / 41, 14 / 48, 15 / 55, 16 / 62, 27 / 104]

path condition with constraints:
(((i$^{82}_{110}$ < len$^{82}_{2}$) & (x$^{87}_{62}$[i$^{87}_{110}$] < x$^{87}_{62}$[result$^{87}_{108}$]) & (i$^{87}_{110}$ < len$^{87}_{2}$) & ((4 = i$^{82}_{110}$) | (4 =
    result$^{82}_{108}$))) | (4 = result$^{104}_{108}$)) & (a < high) & (b < high) & (c < high) & (d <
    high) & (e < high) & (f < high) & (g < high) & (h < high) & (i$^{82}_{110}$ $\neq$ 4)

 = iteration 129 =
No more satisfying solutions. No flow.

=======
```

Figure A.16: Program Dependence Graph of Min



```
formula length: 125027
iterations: 129
time used: 158.971s
thereof time for SMT solver: 153.996s
```

B Used Intermediate Representation

B.1 Used Language

In our descriptions we use an imperative subset of Java containing integers, while loops and arrays. To avoid parsing the Java source code by our self we use Java bytecode with debug information and make some assumptions:

Assumption 4. *1. Array references are final and only assigned to new arrays. This prevents ambiguities and aliasing, therefore we don't need to perform a point to analysis and arrays can be referred by their variable names.*

2. Only a subset of bytecode operations is used.

B.2 Basic Blocks

For the further assumptions we need the concept of basic blocks. Basic blocks are blocks of code, which have a single entry and a single exit point. They are used in compilers. (P.a. see [11])

We call a basic block closed over the stack iff for all valid executions of the Java bytecode the stack size at the beginning and the end of the basic block is zero.

A basic block is closed and minimal over the stack iff it is closed and after each operation without the last the stack size is not zero.

From now on, if we use basic block we mean a basic block, which is minimal and closed over the stack.

Assumption 5. *The code can be divided into basic blocks.*

Let $\mathcal{B}$ be the set of basic blocks.

Assumption 6. *1. $\forall b \in \mathcal{B}$: the last operation of b is either an assignment, an unconditional jump, a conditional jump or a return statement.*

2. $\forall b \in \mathcal{B}$: each operation of b without the last is not an assignment.

Given assumption 6 we can categorize basic blocks by their task: assigning, branching, jumping and returning basic blocks. For a branching basic block $b \in \mathcal{B}$ we can compute the branch condition $c(b)$.

From assumption 5 and 6 follows that the only information that can leave a basic block is either an influence to the control flow or a unique variable which is set.

This intermediate representation of the code allows a simple construction of SSA form: The number of the last byte code operation provides the SSA subscript, as it is unique. The same scheme is used for the unique number identifying a CFG node.