Sparse Analysis of Variable Path Predicates Based Upon SSA-Form

Thomas S. Heinze and Wolfram Amme

Institute of Computer Science, Friedrich Schiller University Jena, Germany
{t.heinze,wolfram.amme}@uni-jena.de

Abstract. Static Single Assignment Form benefits data flow analysis by its static guarantees on the definitions and uses of variables. In this paper, we show how to exploit these guarantees to enable a sparse data flow analysis of variable predicates, for gaining a rich predicate-based and path-oriented characterization of a program's variables' values.

1 Introduction

Static Single Assignment Form (SSA-form) [6] is now widely used as an intermediate representation for supporting program analysis and optimization. Various analysis and optimization techniques have been defined for SSA-form, each exploiting the properties of SSA-form to enable sparse analysis. In a *sparse data flow analysis*, instead of propagating abstract information about global program state along the program's control flow, as done in classical data flow analysis [15], information is propagated only from the information source to the points where the information is needed, in case of SSA-form therefore from variable definition to variable use. As a result, less information is stored at fewer program points.

In this paper, we enlarge the set of sparse analyses on SSA-form by a novel technique for deriving *variable predicates* as a predicate-based abstraction on a program's variables' values. We will in particular show, how the static single assignment property of SSA-form naturally facilitates the analysis sparseness, in that it allows for the characterization of variables' values by predicates collected along the chains of data dependences for the variables' defining instructions, instead of using global program state. Furthermore, it enables the incorporation and derivation of path information in terms of set-based predicate encodings, in order to distinguish variables' values among different control flow paths. As variable predicates in this way constitute a rich though finite model for a program's variables, beside others, methods for program verification and model checking can benefit from incorporating the information derived by our analysis.

The rest of the paper is structured as follows: Sect. 2 introduces foundations and main concepts, namely SSA-form, variable predicates and our sparse analysis for deriving them. In Sect. 3, we map the analysis on to the notion of montone dataflow frameworks for proving its correctness. Afterwards, improvements are developed with respect to spurious data flow and unreachable code. Sect. 5 sketches the use of the analysis in a system for the generation of more precise low-level models of WS-BPEL programs used for model checking. Eventually, the paper summarizes and relates our work in Sect. 6 and Sect. 7, respectively.

2 Sparse Analysis of Variables Path Predicates

We are interested in deriving a predicate-based characterization of variables' values. In other words, a predicate for each program variable, describing the fraction of program state relevant for its value. In addition, the derived predicates shall distinguish among a variable's value on differing control flow paths. A natural match for the thus defined analysis problem is *Static Single Assignment Form (SSA-form)* [6], as it provides for the predicates to describe program state due to its single assignment property, and at the same time separates the variable definitions of different control flow paths. Beyond that, SSA-form supports a sparse analysis, which enables a more scalable derivation of predicates. Therefore, we will first introduce SSA-form and preliminaries in the following section. After that, we formalize our concept of predicate-based characterization of variables' values by so-called *variable (path) predicates*, which allows us then to define the derivation of a program's variable predicates using sparse data flow analysis.

2.1 Program Representation in SSA-Form

We represent a program in terms of a control flow graph, i.e., a directed graph $CFG = (N, E)$ with a set of nodes N, a set of edges $E \subseteq N \times N$, unique start node $s \in N$, and unique end node $e \in N$. A (control flow) path from $n_0 \in N$ to $n_k \in N$ is a sequence $(n_0, n_1), (n_1, n_2), \ldots, (n_{k-1}, n_k)$ such that $(n_i, n_{i+1}) \in E$ for $i < k$. We assume that every node is on a path from the start to the end node. Nodes are labeled by $i \in \mathit{Instr}$ or $c \in \mathit{Cond}$, where Instr is the set of the program's instructions and Cond the set of the program's branching conditions, respectively. Instructions and branching conditions are defined over the program's variables Var. Furthermore, let $\overline{\mathit{Pred}_x}$ denote the set of instructions and branching conditions, where x is not used nor defined.

Our analysis operates on *SSA-form* [6], which guarantees for each variable a statically unique definition. This way, variables behave like values, which also means that the relation (def-use chain) between the instruction defining variable x, denoted $\mathit{def}(x) \in \mathit{Instr}$, and the set of instructions where x is used, denoted $\mathit{uses}(x) \subseteq \mathit{Instr}$, is implicitly given. For convenience, let $\mathit{var}(i)$ denote the variable defined by $i \in \mathit{Instr}$ and node denote the node $n \in N$ for $x \in \mathit{Var}$, where n's label links to $\mathit{def}(x)$. As usual, SSA-form is realized by introducing a new variable for each static definition and renaming uses accordingly. At join nodes of the control flow graph, i.e., nodes with mupltiple incoming edges, Φ-functions $x = \Phi(x_1, \ldots, x_n)$ are inserted to merge confluent variable definitions, such that the value of x equals x_j if the join node is reached via its j-th incoming edge.

A trick we use for incorporating the effects of a branching condition into SSA-form and thus to support analysis is to insert assertions at the true and false branch. Without loss of generality, we assume branching conditions $x \mathit{\ op\ } c$, where x is a variable and c a constant. An assertion then looks like $y = \mathit{assert}(x \mathit{\ op\ } c)$, where $x \mathit{\ op\ } c$ is the branching predicate valid at the respective branch. Uses of x inside the branch are updated to the newly defined variable y, whose value

equals x but is guaranteed to satisfy the branching predicate. For the presentation of our sparse analysis of variable predicates, we will focus – without loss of generality, on scalar assignment expressions and neglect, e.g., memory operations or compositve data structures. However, note that these constructs can be addressed by basic extensions [6], for example introducing a virtual variable for heap storage, or by using tailored variants of SSA-form like HSSA-form [3].

Example 1. As an example, consider the following program snippet (left), its SSA-form (middle), and its SSA-form with assertions added (right):

```
x = 1;
while (x % 2) {
   x = x + 2;
}
```

```
x1 = 1;
x2 = Φ(x1,x3);
while (x2 % 2) {
   x3 = x2 + 2;
}
```

```
x1 = 1;
x2 = Φ(x1,x4);
while (x2 % 2) {
   x3 = assert(x2 % 2);
   x4 = x3 + 2;
}
```

2.2 Variable Predicates

For the formal development of the analysis, we next introduce *atomic predicates*, *variable path predicates*, and *all-paths variable predicates*. Apparently, we can interpret instructions and branching conditions as first-order predicates characterizing variables' values, when substituting the assignment with the equality operator. For instance, the value of variable `x`, defined through the constant assignment `x = 10`, can be described using equality predicate $x = 10$. A program's instructions and branching conditions thus constitute the set of atomic predicates, augmented with additional simple equalities of form $x = y$, $x, y \in \mathit{Var}$:

Definition 1. *Let Instr, Cond, and Var denote the set of instructions, branching conditions, and variables of the given program, respectively. The set of* atomic predicates *is defined by* $\mathit{Pred} = \mathit{Instr} \cup \mathit{Cond} \cup \{x = y \mid x, y \in \mathit{Var}\}$.

Characterizing a variable's value for a single path is done by a *variable path predicate*, i.e., a conjunction of instructions and branching conditions determining the variable's value on this path. We denote such a conjunction as a set of atomic predicates. Considering instructions `x = 10; y = x * 2`, we thus get $\{x = 10, y = x * 2\}$, representing the conjunction $x = 10 \wedge y = x * 2$ for describing `y`'s value. Note that we, as usual, universally quantify over free variables. In order to reflect a variable's value for all paths, the variable path predicates for individual paths are disjunctively combined. The resulting formulæ, called *all-paths variable predicate* (or simply *variable predicate*), is also denoted as set:

Definition 2. *A* variable path predicate *is a set* $p \in \mathcal{P}(\mathit{Pred})$ *interpreted as conjunction of atomic predicates. An* (all-paths) variable predicate *is then a set* $f \in \mathcal{P}(\mathcal{P}(\mathit{Pred}))$ *interpreted as disjunction of conjunctions of predicates.*

Assuming, e.g., `if (a < 0) x = 1; else x = 3; y = x`, variable `y`'s value can be characterized by variable predicate $\{\{x = 1, y = x\}, \{x = 3, y = x\}\}$, where `y`'s path predicates for the branching's true and false branch are disjunctively combined by means of formulæ $(x = 1 \wedge y = x) \vee (x = 3 \wedge y = x)$.

Eventually, we define the empty set $\emptyset$ to denote the truth value *true*. This is justified by variable predicates acting as premises about variables' value, where *true* is the weakest and therefore always safe premise.

2.3 Derivation of Variable Predicates

As mentioned before, a program's SSA-form allows for a sparse derivation of variable predicates. To this end, instead of propagating a set of predicates, denoting the program state for all variables, along all control flow paths, we derive variable predicates by analyzing each variables' definitions and uses. In principle, there are two reasons why this works: First, SSA-form guarantees that each variable is (statically) defined once and, with the exception of Φ-functions, every use is dominated by its definition. In consequence, a variable's value does not change along the paths from definition to use, such that the program state valid directly after the definition can be used to characterize the variable's value on all paths. Second, only part of the overall program state is relevant for a single variable, which can in particular be captured by following the variable's data dependences along the def-use chains directly encoded in SSA-form. Note that this way, we may omit predicates affecting a variable's value through side effects or control dependences, which though does not invalidate the approach as it is always safe to infer a weaker variable predicate, i.e., p instead of $p \wedge q$.

Thus, we assign each variable x its variable predicate $pred(x)$ based upon its defining instruction. For a constant assignment $x = c$, we can obviously set $pred(x) = \{\{x = c\}\}$. In case of a simple assignment $x = y$, the union over variable path predicates $p \in pred(y)$, each augmented by the equality predicate $x = y$, is used, in this way including derived information about y also in x's variable predicate. However, in order to prevent inconsistent predicates in case of cyclic data dependences, atomic predicates other than $x = y$ containing variable x are removed. The same principle applies to an assertion $x = assert(y\ op\ c)$, though the asserted predicate $x\ op\ c$ for variable x is added to each path predicate besides the equality predicate $x = y$. The variable predicate for a variable defined through assignment $x = y\ op\ z$ is derived by considering all pairs (p, q) of variable path predicates $p \in pred(y)$, $q \in pred(z)$ of the operand variables, flattening each into a single set $p \cup q$ and adding the assignment as predicate. Once more, existing atomic predicates containing variable x are removed to avoid inconsistencies. Remember that due to SSA-form, x's definition is dominated by the definitions of variables y and z, which means that predicates for y and z are also reasonable for depicting the program state at the assignment. Consider, e.g., `x = y * z` with $pred(y) = \{\{y = 10\}\}$, $pred(z) = \{\{z = v, v = -1\}, \{z = w, w = 1\}\}$, then $pred(x) = \{\{z = v, v = -1, y = 10, x = y * z\}, \{z = w, w = 1, y = 10, x = y * z\}\}$. Finally, in case of a Φ-function $x = \Phi(x_1, \ldots, x_n)$ merging confluent definitions x_j into a single value x, the variable predicate equals the union of the operands path predicates, each augmented with an equality predicate $x = x_j$ for x and the respective operand x_j. Atomic predicates containing a variable simultaneously defined with `x`, i.e., within the same control flow graph node, are again removed. As an example, assuming `x = Φ(x1,x2)` with predicates $pred(x_1) = \{\{x_1 = 10\}\}$

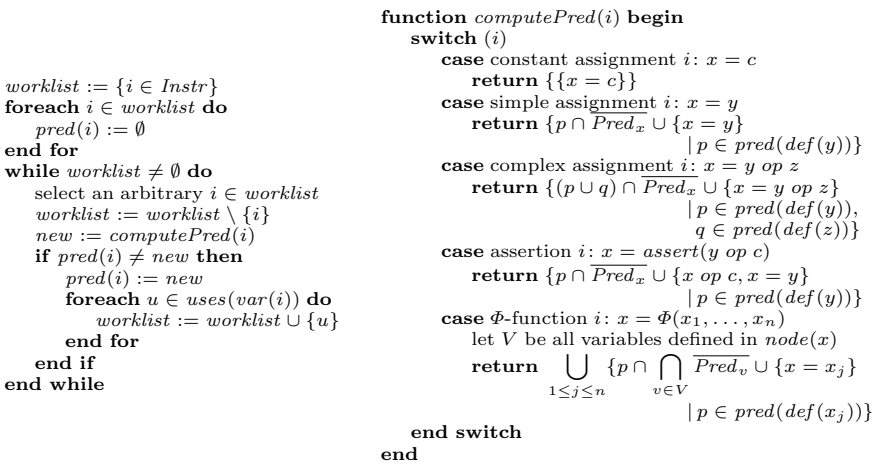

Fig. 1. Sparse analysis of variable predicates

and $pred(x_2) = \{\{x_2 = z + 3, z = 9\}\}$, then $pred(x) = \{\{x_1 = 10, x = x_1\}, \{x_2 = z+3, z = 9, x = x_2\}\}$. The following definition summarizes these rules for the derivation of a program's variable predicates:

Definition 3. *Variable predicates are defined for a given program in SSA-form by* $pred\colon Var \to \mathcal{P}(\mathcal{P}(Pred))$ *for each variable* $x \in Var$ *according to its defining instruction* $i \in Instr$ *based upon the following equations:*

- *constant assignment* $i\colon x = c$
  $pred(x) = \{\{x = c\}\}$
- *simple assginment* $i\colon x = y$
  $pred(x) = \{p \cap \overline{Pred_x} \cup \{x = y\} \,|\, p \in pred(y)\}$
- *complex assignment* $i\colon x = y\ op\ z$
  $pred(x) = \{(p \cup q) \cap \overline{Pred_x} \cup \{x = y\ op\ z\} \,|\, p \in pred(y), q \in pred(z)\}$
- *assertion* $i\colon x = assert(y\ op\ c)$
  $pred(x) = \{p \cap \overline{Pred_x} \cup \{x = y, x\ op\ c\} \,|\, p \in pred(y)\}$
- *Φ-function* $i\colon x = \Phi(x_1, \ldots, x_n)$, *with* $V \subseteq Var$ *variables defined in* $node(x)$
  $$pred(x) = \bigcup_{1 \le j \le n} \{p \cap \bigcap_{v \in V} \overline{Pred_v} \cup \{x = x_j\} \,|\, p \in pred(x_j)\}$$

For solving the equation system defined by Definition 3, we use the worklist algorithm in Fig. 1. The algorithm computes variable predicates by *computePred* based upon the variables' definitions as described above. Due to SSA-form's single assignment property, variables can be identified by their unique defining instructions, such that variable predicates *pred* are assigned to instructions instead of variables. Having initialized all variable predicates to the empty set and the worklist to comprise the program's instructions, the algorithm continuously

takes an instruction i from the worklist and recomputes its variable predicate using *computePred*. Each time a change in its value is observed, $pred(i)$ is updated accordingly and all use sites of the variable defined by i are again added to the worklist. If eventually a stable solution is reached, the algorithm terminates.

Reconsidering Example 1 and applying the algorithm, we get the solution:

$$
\begin{aligned}
pred(x_1) &= \{\{x_1 = 1\}\} \\
pred(x_2) &= \{\{x_1 = 1, x_2 = x_1\}, \{x_1 = 1, x_3 \,\%\, 2, x_4 = x_3 + 2, x_2 = x_4\}\} \\
pred(x_3) &= \{\{x_1 = 1, x_2 = x_1, x_3 \,\%\, 2, x_3 = x_2\}, \{x_1 = 1, x_2 = x_4, x_3 \,\%\, 2, x_3 = x_2\}\} \\
pred(x_4) &= \{\{x_1 = 1, x_2 = x_1, x_3 \,\%\, 2, x_3 = x_2, x_4 = x_3 + 2\}, \\
&\qquad \{x_1 = 1, x_3 \,\%\, 2, x_3 = x_2, x_4 = x_3 + 2\}\}
\end{aligned}
$$

As can be seen, the derived predicate for variable $\mathtt{x}_2$ consists of two path predicates, characterizing $\mathtt{x}_2$'s value before initially entering the loop and before reentering the loop, respectively. Note that, assuming C semantics such that $x \,\%\, 2$ equals $x \,\%\, 2 \neq 0$, $\mathtt{x}_2$'s predicate apparently determines the loop condition's value, i.e., $(x_1 = 1 \wedge x_2 = x_1) \vee (x_1 = 1 \wedge x_3 \,\%\, 2 \neq 0, x_4 = x_3 + 2, x_2 = x_4) \models x_2 \,\%\, 2 \neq 0$, as can be automatically inferred using a SMT solver for testing the implication.

3 Correctness of the Analysis Algorithm

In the previous section, we have developed a sparse analysis algorithm for the derivation of variable predicates. In this section, we will prove that the algorithm always terminates while yielding the correct set of predicates for characterizing variables' values. The algorithm can be seen as an optimized version of the general iterative algorithm for data flow problems [15], which is already proven to terminate with a safe solution for monotone problems. We will therefore first present the concept of a *monotone data flow framework* and afterwards show, how our algorithm and conceptual universe can be mapped onto a monotone data flow framework for proving the correctness of the sparse analysis algorithm.

The principle of data flow analysis is to gather information for each instruction by iteratively propagating locally computed data flow information through the control flow graph of a program. In general, each data flow problem can be modeled using a *data flow framework* $(L, \wedge, F)$, where L is the *data flow information set*, $\wedge$ is the *meet operator*, and F is the set of *semantic functions*. The data flow information set is a conceptual universe of objects upon which the analysis is working. A semantic function corresponds to an instruction and models the effect that an execution of the instruction has onto the incoming information. The meet operator implements the effect of joining control flow paths. A maximum fixpoint solution for a data flow framework can be computed, if and only if the semantic functions are monotone and $(L, \wedge)$ forms a bounded semi lattice with a one element 1 and a zero element 0 [15].[1] Data flow frameworks satisfying these requirements are called *monotone data flow frameworks (MDF)*.

[1] For the general iterative algorithm, we refer the reader to [15] or Sect. A.

3.1 Data Flow Analysis of Variable Predicates

There are multiple ways for constructing a MDF for deriving variable predicates. An obvious approach is to derive for each node n of a control flow graph a set of pairs (x, p), in which x stands for a variable and p for a set of predicates that characterize the value of x at node n. While a single pair (x, p) describes a path predicate of x, the union of all pairs represents x's all-paths variable predicate. The data flow framework for the calculation of variable predicates is defined by $MDF_{\mathrm{VP}} = (L_{\mathrm{VP}}, \wedge_{\mathrm{VP}}, F_{\mathrm{VP}})$. where $L_{\mathrm{VP}} = \mathcal{P}(\mathit{Var} \times \mathcal{P}(\mathit{Pred}))$ and $\wedge_{\mathrm{VP}} \colon L \to L$ is the set-theoretic union operator such that $l \wedge_{\mathrm{VP}} k = l \cup k$ for all $l, k \in L$.

Lemma 1. *$(L_{\mathrm{VP}}, \wedge_{\mathrm{VP}})$ is a bounded semi lattice with zero element $0 \in L_{\mathrm{VP}}$ and one element $1 \in L_{\mathrm{VP}}$ such that $\forall l \in L_{\mathrm{VP}} \colon\ l \wedge_{\mathrm{VP}} 1 = l$ and $l \wedge_{\mathrm{VP}} 0 = 0$.*

Proof. Since *Var* and *Instr* are finite sets for a given program, and thus is $\mathcal{P}(\mathit{Var} \times \mathcal{P}(\mathit{Pred}))$, the lemma follows immediately from the fact that for every finite set M, $(\mathcal{P}(M), \cup)$ is a bounded semi lattice with $1 = \emptyset$ and $0 = M$. □

In the control flow graph used for the analysis, each node stands for a uniquely labeled SSA instruction. Therefore we can unambiguously assign a semantic function to each of the nodes. This semantic function will be used to transform the set of variable path predicates when processing the node. Each semantic function models for a given node of the control flow graph the effect of executing the node, i.e., the node's attached instructions on the incoming data flow information.

Definition 4. *The semantic functions F_{VP} are defined according to $i \in \mathit{Instr}$:*

- *Φ-function $i \colon x = \Phi(x_1, \ldots, x_n)$, with $V \subseteq \mathit{Var}$ variables defined in node(x)*
  $VP_{\mathrm{out}} = \mathit{update}_i(\mathit{remove}(VP_{\mathrm{in}}, V))$
- *any other instruction i defining value x*
  $VP_{\mathrm{out}} = \mathit{update}_i(\mathit{remove}(VP_{\mathrm{in}}, \{x\}))$

where $\mathit{remove} \colon L_{\mathrm{VP}} \times \mathcal{P}(\mathit{Var}) \to L_{\mathrm{VP}}$ is defined for $k \in L_{\mathrm{VP}}$ and $V \subseteq \mathit{Var}$ by

- $\mathit{remove}(k, V) = \{(y, p \cap \bigcap_{v \in V} \overline{\mathit{Pred}_v} \mid (y, p) \in s \wedge y \notin V\}$

and $\mathit{update}_i \colon L_{\mathrm{VP}} \to L_{\mathrm{VP}}$ is defined for $l \in L_{\mathrm{VP}}$ according to $i \in \mathit{Instr}$ by:

- *constant assignment $i \colon x = c$*
  $\mathit{update}_i(l) = l \cup \{(x, \{x = c\})\}$
- *simple assignment $i \colon x = y$*
  $\mathit{update}_i(l) = l \cup \{(x, p \cup \{x = y\}) \mid (y, p) \in l\}$
- *complex assignment $i \colon x = y\ \mathit{op}\ z$*
  $\mathit{update}_i(l) = l \cup \{(x, p \cup q \cup \{x = y\ \mathit{op}\ z\}) \mid (y, p), (z, q) \in l\}$
- *assertion $i \colon x = \mathit{assert}(y\ \mathit{op}\ c)$*
  $\mathit{update}_i(l) = l \cup \{(x, p \cup \{x = y, x\ \mathit{op}\ c\}) \mid (y, p) \in l\}$

– *Φ-function* $i\colon x = \Phi(x_1, \ldots, x_n)$
$update_i(l) = l \cup \bigcup_{1 \le j \le n} \{(x, p \cup \{x = x_j\}) \mid (x_j, p) \in l\}$

Lemma 2. *The semantic functions* F_{VP} *defined in Definition 4 are monotone.*

Proof. To prove that the semantic functions $f \in F_{\mathrm{VP}}$ are monotone, we have to show $l \sqsubseteq_{\mathrm{VP}} k \Rightarrow f(l) \sqsubseteq_{\mathrm{VP}} f(k)$ for every $l, k \in L_{\mathrm{VP}}$. Since $l \sqsubseteq_{\mathrm{VP}} k$ iff $l = l \wedge_{\mathrm{VP}} k$ and $\wedge_{VP} = \cup$, we can alternatively show that $f(l) \cup f(k) \subseteq f(l \cup k)$ holds. First, we prove $remove(l, V) \cup remove(k, V) = remove(l \cup k, V)$ for $l, k \in L_{\mathrm{VP}}$:

– $remove(l, V) \cup remove(k, V)$
$= \{(y, p \cap \bigcap_{v \in V} \overline{Pred_v} \mid (y, p) \in l \wedge y \notin V\} \cup \{(y, p \cap \bigcap_{v \in V} \overline{Pred_v} \mid (y, p) \in k \wedge y \notin V\}$
$= \{(y, p \cap \bigcap_{v \in V} \overline{Pred_v} \mid (y, p) \in l \cup k \wedge y \notin V\} = remove(l \cup k, V)$

and thereafter that $update_i(l) \cup update_i(k) \subseteq update_i(l \cup k)$ for every $i \in Instr$:

– constant assignment $i\colon x = c$
$update_i(l) \cup update_i(k) = l \cup \{(x, \{x = c\})\} \cup k \cup \{(x, \{x = c\})\}$
$= l \cup k \cup \{(x, \{x = c\})\} = update_i(l \cup k)$
– simple assignment $i\colon x = y$
$update_i(l) \cup update_i(k) = l \cup \{(x, p \cup \{x = y\}) \mid (y, p) \in l\}$
$\cup\, k \cup \{(x, p \cup \{x = y\}) \mid (y, p) \in k\}$
$= l \cup k \cup \{(x, p \cup \{x = y\}) \mid (y, p) \in l \cup k\} = update_i(l \cup k)$
– complex assignment $i\colon x = y\ op\ z$
$update_i(l) \cup update_i(k) = l \cup \{(x, p \cup q \cup \{x = y\ op\ z\}) \mid (y, p), (z, q) \in l\}$
$\cup\, k \cup \{(x, p \cup q \cup \{x = y\ op\ z\}) \mid (y, p), (z, q) \in k\}$
$\subseteq l \cup k \cup \{(x, p \cup q \cup \{x = y\ op\ z\}) \mid (y, p), (z, q) \in l \cup k\} = update_i(l \cup k)$
– assertion $i\colon x = assert(y\ op\ c)$
$update_i(l) \cup update_i(k) = l \cup \{(x, p \cup \{x = y, x\ op\ c\}) \mid (y, p) \in l\}$
$\cup\, k \cup \{(x, p \cup \{x = y, x\ op\ c\}) \mid (y, p) \in k\}$
$= l \cup k \cup \{(x, p \cup \{x = y, x\ op\ c\}) \mid (y, p) \in l \cup k\} = update_i(l \cup k)$
– Φ-function $i\colon x = \Phi(x_1, \ldots, x_n)$
$update_i(l) \cup update_i(k) = l \cup \bigcup_{1 \le j \le n} \{(x, p \cup \{x = x_j\}) \mid (x_j, p) \in l\}$
$\cup\, k \cup \bigcup_{1 \le j \le n} \{(x, p \cup \{x = x_j\}) \mid (x_j, p) \in k\}$
$= l \cup k \cup \bigcup_{1 \le j \le n} \{(x, p \cup \{x = x_j\}) \mid (x_j, p) \in l \cup k\} = update_i(l \cup k)$

From the fact that the composition of the thus monotone functions *remove* (with respect to its first argument) and $update_i$ is monotone, follows the lemma. □

Theorem 1. *The general iterative algorithm terminates with the maximum fixpoint solution for each instance of the data flow framework* MDF_{VP}.

Proof. This is an immediate consequence of MDF_{VP} being a monotone data flow framework according to Lemma 1 and Lemma 2. □

In fact, the algorithm for sparse analysis of variable predicates we have presented in Sect. 2 can be seen as an optimized variant of the general iterative algorithm solving MDF_{VP}. In principle, the sparse analysis differs in that path predicates for all variables are not propagated along the control flow, as is done by the general iterative algorithm, but rather derived and stored directly at the control flow graphs variable-defining instructions in terms of variable predicates. Since each variable is statically defined once in SSA-form and MDF_{VP}'s merge operator is the set union, the sparse approach does not invalidate the correctness of the analysis, which allows us to state the following corollary:

Corollary 1. *A safe solution to the equation system pred defined in Definition 3 can be computed using the fixpoint algorithm in Fig. 1.*

4 Improvements of the Analysis

Due to the nature of sparse analysis, precision of our analysis of variable predicates is impeded by the omittance of program information which is not represented by the relations of variable definition and use. However, two reasons for imprecision, namely unreachable code and the local merging of data flow facts at join nodes, can be addressed by the analysis extensions described next.

4.1 Spurious Data Flow

The presented sparse analysis exploits the single-assignment property of SSA-Form and propagates data flow information, i.e., variable predicates, only along def-use chains. While this apparently benefits analysis performance, precision is being lost at join nodes, since the analysis does not track the correlation of variables' values defined conjointly along converging control flow paths.

Example 2. Consider the program snippet below and its inferred predicates:

```
if (...) {
   a1 = 2;
   b1 = 3;
} else {
   a2 = 3;
   b2 = 2;
}
a3 = Φ(a1,a2);
b3 = Φ(b1,b2);
c1 = a3 + b3;
```

$$pred(a_1) = \{\{a_1 = 2\}\} \qquad pred(b_1) = \{\{b_1 = 3\}\}$$
$$pred(a_2) = \{\{a_2 = 3\}\} \qquad pred(b_2) = \{\{b_2 = 2\}\}$$
$$pred(a_3) = \{\{a_1 = 2, a_3 = a_1\}, \{a_2 = 3, a_3 = a_2\}\}$$
$$pred(b_3) = \{\{b_1 = 3, b_3 = b_1\}, \{b_2 = 2, b_3 = b_2\}\}$$
$$\begin{aligned} pred(c_1) = \{&\{a_1 = 2, a_3 = a_1, b_1 = 3, b_3 = b_1, c_1 = a_3 + b_3\},\\ &\{a_1 = 2, a_3 = a_1, b_2 = 2, b_3 = b_2, c_1 = a_3 + b_3\},\\ &\{a_2 = 3, a_3 = a_2, b_1 = 3, b_3 = b_1, c_1 = a_3 + b_3\},\\ &\{a_2 = 3, a_3 = a_2, b_2 = 2, b_3 = b_2, c_1 = a_3 + b_3\}\} \end{aligned}$$

Therein, the values of $\mathtt{a}_3$ and $\mathtt{b}_3$ are characterized with different path predicates, so that the values on the true and false branch are distinguished. Though after the join, when considering $\mathtt{c}_1$ = $\mathtt{a}_3$ + $\mathtt{b}_3$, spurious combinations of path predicates arise, e.g., $\{a_1 = 2, a_3 = a_1, b_2 = 2, b_3 = b_2, c_1 = a_3 + b_3\}$, coming from mutually exclusive control flow paths. Therefore, $\mathtt{c}_1$'s inferred variable predicate is imprecise in that it allows for values 4, 5, 6, while only 5 can occur at runtime.

In order to remove this imprecision but still support a sparse analysis, variable path predicate are attached information about the represented control flow paths. To this end, we introduce *path designators* for denoting a path based on the edges entering a control flow graph's join nodes throughout the path:

Definition 5. *A* path designator $\delta \in \mathcal{P}(N \times \mathbb{N})$ *is a definite relation, such that* $\forall n \in N\colon (n,i) \in \delta \wedge (n,j) \in \delta \rightarrow j = j$*, which determines for each node* n *of a subset of a control flow graph's join nodes a predecessor node using the predecessor's index. Overriding of a path designator* δ *by a path designator* γ *is defined as* $\delta \oplus \gamma = \{(x,i) \mid (x,i) \in \delta \wedge \nexists (x,j) \in \gamma\} \cup \{(y,i) \mid (y,i) \in \gamma\}$.

Apparently, two path designators δ and γ defining different predecessors $\delta(n) \neq \gamma(n)$ for the same node n characterize mutually exclusive control flow paths. Augmenting variable path predicates with path designators, we are able to rule out the spurious combinations of path predicates:

Definition 6. Variable predicates with path designators *are defined for a given control flow graph by* $pred^*\colon Instr \rightarrow \mathcal{P}(\mathcal{P}(N \times \mathbb{N}) \times \mathcal{P}(Pred))$ *for each variable* $x \in Var$ *according to its defining instruction* $i \in Instr$ *based upon equations:*

- *constant assignment* $i\colon x = c$
  $pred^*(x) = \{(\emptyset, \{x = c\})\}$
- *simple assginment* $i\colon x = y$
  $pred^*(x) = \{(\delta, p \cap \overline{Pred_x} \cup \{x = y\}) \mid (\delta, p) \in pred^*(y)\}$
- *assertion* $i\colon x = assert(y\ op\ c)$
  $pred^*(x) = \{(\delta, p \cap \overline{Pred_x} \cup \{x = y, x\ op\ c\}) \mid (\delta, p) \in pred^*(y)\}$
- *Φ-function* $x\colon x = \Phi(x_1, \ldots, x_n)$*, with* $V \subseteq Var$ *variables defined in node*(x)
  $$pred^*(x) = \bigcup_{1 \leq j \leq n} \{(\delta \oplus \{(node(x), j)\}, p \cap \bigcap_{v \in V} \overline{Pred_v} \cup \{x = x_j\}) \mid (\delta, p) \in pred^*(x_j)\}$$
- *complex assignment* $i\colon x = y\ op\ z$
  $$pred^*(x) = \{(\delta \cup \gamma, (p \cup q) \cap \overline{Pred_x} \cup \{x = y\ op\ z\}) \mid (\delta, p) \in pred^*(y), (\gamma, q) \in pred^*(z) \wedge (\delta \cup \gamma)\ \textit{is definite}\}$$

For constant assignments, we thus attach the empty set as path designator to derived path predicates. In case of simple assignments and assertions, path designators are merely propagated as there is just a single variable operand and line of control. For a Φ-function, path designators attached to the operands'

path predicates are updated according to the operands' indices, determining the respective predecessors of the Φ-function's join node. In case of a complex assignment, the union of path designators is created for each combination of the operands' path predicates. If the union is not a definite relation, the path predicates come from mutually exclusive paths and their combination is skipped.

In order to solve the equation system of Definition 6, we can again use a variant of the algorithm in Fig. 1. Reconsidering Example 2 and assuming a node *join* for the two Φ-functions $\texttt{a}_3 = \Phi(\texttt{a}_1,\texttt{a}_2)$ and $\texttt{b}_3 = \Phi(\texttt{b}_1,\texttt{b}_2)$, we get:

$$pred^*(a_1) = \{(\emptyset, \{a_1 = 2\})\} \qquad pred^*(a_2) = \{(\emptyset, \{a_2 = 3\})\}$$
$$pred^*(b_1) = \{(\emptyset, \{b_1 = 3\})\} \qquad pred^*(b_2) = \{(\emptyset, \{b_2 = 2\})\}$$
$$pred^*(a_3) = \{(\{(join, 1)\}, \{a_1 = 2, a_3 = a_1\}), (\{(join, 2)\}, \{a_2 = 3, a_3 = a_2\})\}$$
$$pred^*(b_3) = \{(\{(join, 1)\}, \{b_1 = 3, b_3 = b_1\}), (\{(join, 2)\}, \{b_2 = 2, b_3 = b_2\})\}$$
$$pred^*(c_1) = \{(\{(join, 1)\}, \{a_1 = 2, a_3 = a_1, b_1 = 3, b_3 = b_1, c_1 = a_3 + b_3\}),$$
$$(\{(join, 2)\}, \{a_2 = 3, a_3 = a_2, b_2 = 2, b_3 = b_2, c_1 = a_3 + b_3\})\}$$

Therein, each path predicate has a conjoined path designator, determining the represented control flow path in terms of *join*'s predecessors. For instance, $\texttt{a}_3$'s path predicates $\{a_1 = 2, a_3 = a_1\}$, $\{a_2 = 3, a_3 = a_2\}$ are assigned designators $\{(join, 1)\}$, $\{(join, 2)\}$, denoting the if and else branch, respectively. Spurious combinations of path predicates are thus ruled out for variable $\texttt{c}_1 = \texttt{a}_3 + \texttt{b}_3$ such that $\texttt{c}_1$'s all-paths predicate allows for deriving its precise value 5.

Eventually, we can state a correctness argument similar to the one in Sect. 3 for the thus defined improved analysis with path designators:

Corollary 2. *A safe solution to the equation system $pred^*$ defined in Definition 6 can be computed using the fixpoint algorithm in Fig. 1.* [2]

4.2 Unreachable Code

Another source of imprecision is reasoned by *unreachable code*, i.e., program statements that can never be executed due to unsatisfiable branching conditions. The analysis considers data dependences for propagating data flow information, but ignores control dependences, assuming that each branching condition is satisfiable and therefore every branch of the control flow can be executed.

Example 3. Consider the program snippet below and its inferred predicates:

```
if (a1 > 10) {
  a2 = assert(a1 > 10);
  if (a2 > 5) {
    a3 = assert(a2 > 5);
    b1 = 1;
  } else {
    a4 = assert(a2 ≤ 5);
    b2 = -1;
  }
  b3 = Φ(b1,b2);
```

$$pred(a_2) = \{\{\ldots, a_2 > 10, a_2 = a_1\}\}$$
$$pred(a_3) = \{\{\ldots, a_2 > 10, a_2 = a_1, a_3 > 5, a_3 = a_2\}\}$$
$$pred(b_1) = \{\{b_1 = 1\}\}$$
$$pred(a_4) = \{\{\ldots, a_2 > 10, a_2 = a_1, a_4 \leq 5, a_4 = a_2\}\}$$
$$pred(b_2) = \{\{b_2 = -1\}\}$$
$$pred(b_3) = \{\{b_1 = 1, b_3 = b_1\}, \{b_2 = -1, b_3 = b_2\}\}$$

[2] For the proof, see Sect. B

As can be seen, in spite of the fact that the condition of the inner branching is always satisfied and its false branch can therefore never be executed, the analysis considers variable $\mathtt{b}_2$'s value -1, defined inside the false branch, to flow into variable $\mathtt{b}_3$. Thus, $\mathtt{b}_3$'s variable predicate is imprecise in that it contains the path predicate $\{b_2 = -1, b_3 = b_2\}$ and consequently allows for values 1 and -1.

Fortunately, we can resort to the approach of combining analyses to include a kind of *unreachable code elimination* [5, 18]. Principle of the combined analysis is to defer the propagation of data flow information through a node until the node is determined to be executable. Therefore, instructions' variable predicates are not computed in an arbitrary order but rather in conformance with the control flow relation. In addition, a branching instruction's condition is evaluated based upon inferred variable predicates. If evaluation results in a definite value, the executed branch is statically known such that all other, unreachable branches can be ignored by the analysis. Otherwise, if inferred variable predicates do not allow for determining the branching result, all branches are considered instead.

The worklist algorithm in Fig. 2 implements the combined analysis, keeping track of executable nodes using bit map *executable*. As before (refer to Fig. 1), instructions' variable predicates *pred* are continuously computed by *computePred* until a fixpoint has been found, though this time, only for instructions whose nodes are marked executable. The function *evaluateCond*, used in the algorithm for evaluating a branching condition, is generic in the applied solver, in that it allows for a SMT solver as well as for, e.g., a simpler constant evaluation. The solver is used to test whether the inferred variable predicate for x implies the value of a condition expression x *op* c. Though, in order to provide for an overapproximation, *evaluateCond* does not test for condition d itself, but rather for its negation. Thus, if $\neg d$ is shown for $pred(x)$, the condition is determined unsatisfiable. We naturally assume for the solver to guarantee that if $\{p, q\} \models d$ (i.e., $p \vee q \models d$) is shown, $p \models d$ and $q \models d$ can be shown as well for $p, q \in \mathcal{P}(Pred)$.

Reconsidering Example 3 and applying the algorithm, we get the solution:

$$
\begin{array}{ll}
pred(a_2) = \{\{\ldots, a_2 > 10, a_2 = a_1\}\} & pred(a_4) = \emptyset \\
pred(a_3) = \{\{\ldots, a_2 > 10, a_2 = a_1, a_3 > 5, a_3 = a_2\}\} & pred(b_2) = \emptyset \\
pred(b_1) = \{\{b_1 = 1\}\} & pred(b_3) = \{\{b_1 = 1, b_3 = b_1\}\}
\end{array}
$$

As can be seen, the inner branching's false branch has been identified unreachable, assuming the used solver able to show $\{\{\ldots, a_2 > 10, a_2 = a_1\}\} \models a_2 > 5$. As a result, the empty set is inferred for variables defined in the false branch, so that $\mathtt{b}_3$'s variable predicate only comprises one path predicate for the reachable true branch and consequently only allows for deriving $\mathtt{b}_3$'s precise value 1.

For proving monotonicity of the thus defined combined analysis with unreachable code elimination, the presumption for the used solver, that if $\{p, q\} \models d$ is shown, also $p \models d$ and $q \models d$ can be shown for $p, q \in \mathcal{P}(Pred)$, is essential. Based upon this presumption, we can again state the following correctness argument:

```
let s ∈ Instr be the start instruction
executable(node(s)) := true
worklist := {s}
pred(s) := ∅
foreach i ∈ Instr \ {s} do
  executable(node(i)) := false
  pred(i) := ∅
end for
while worklist ≠ ∅ do
  select an arbitrary i ∈ worklist
  worklist := worklist \ {i}
  if executable(node(i)) then
    new := computePred(i)
    if pred(i) ≠ new then
      pred(i) := new
      foreach u ∈ uses(var(i)) do
        worklist := worklist ∪ {u}
      end for
    end if
    if i is a branch instruction then
      let t be i′s successor in true branch
      let f be i′s successor in false branch
      let d be i′s branching condition
      if evaluateCond(d) = true then
        executable(node(t)) := true
        worklist := worklist ∪ {t}
      end if
      if evaluateCond(¬d) = true then
        executable(node(f)) := true
        worklist := worklist ∪ {f}
      end if
    else if i has successor s then
      executable(node(s)) := true
      worklist := worklist ∪ {s}
    end if
  end if
end while
```

```
function computePred(i) begin
  switch (i)
    case constant assignment i: x = c
      return {{x = c}}
    case simple assignment i: x = y
      return {p ∩ Pred_x ∪ {x = y}
                | p ∈ pred(def(y))}
    case complex assignment i: x = y op z
      return {(p ∪ q) ∩ Pred_x ∪ {x = y op z}
                | p ∈ pred(def(y)),
                  q ∈ pred(def(z))}
    case assertion i: x = assert(y op c)
      return {p ∩ Pred_x ∪ {x op c, x = y}
                | p ∈ pred(def(y))}
    case Φ-function i: x = Φ(x_1, ..., x_n)
      let V be all variables defined in node(x)
      return ⋃_{1≤j≤n} {p ∩ ⋂_{v∈V} Pred_v ∪ {x = x_j}
                | p ∈ pred(def(x_j))}
                  ∧ node(x)′s jth predecessor
                      is marked as executable}
  end switch
end

function evaluateCond(d) begin
  let d = x op c
  if pred(x) ⊨ ¬d then
    return false
  else
    return true
  end if
end
```

Fig. 2. Sparse analysis of variable predicates with unreachable code elimination

Corollary 3. *The algorithm in Fig. 2 computes a safe solution to the equation system pred defined in Definition 3 (or to pred* defined in Definition 6), while omitting unreachable code.* [3]

5 Application to Model Checking

We have implemented the presented analysis in a system for the generation of more precise low-level models used for model checking distributed business processes [13]. Fig. 3 contains a sketch of the system. The system expects a program of the *Web Services Business Process Execution Language (WS-BPEL)* [20], i.e., an XML-based industry standard for implementing distributed business processes. A WS-BPEL program is then translated into our SSA-form intermediate format, serving as basis for static analyses and optimizations. Eventually, the intermediate format is transformed into low-level Petri nets, the ordinary for-

[3] For the proof, see Sect. C.

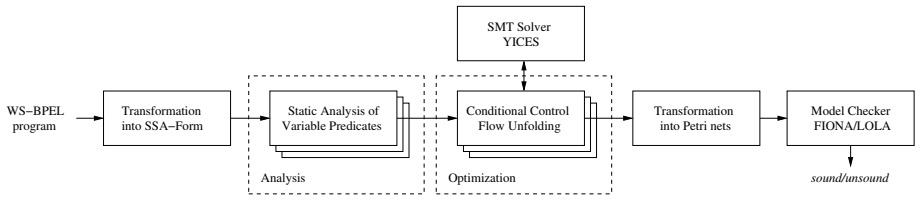

Fig. 3. System for the improved Petri-net-based verification of WS-BPEL programs

malism in the area of business process verification, which are afterwards passed into the model checker *Fiona/LoLA*[4] for verifiying critical soundness properties.

Program data is usually omitted when compiling Petri net models, which however impairs precision when using the models for verification. Integrating program data to regain precision though requires some kind of data abstraction, i.e., a finite model for program data. We have thus used variable path predicates, as derived for a WS-BPEL program by our analysis, to encode program data into the low-level models by means of a technique called *control flow unfolding*. This technique in principle splits and duplicates control flow for paths revealing distinct variable path predicates. As a side effect, a program's branching conditions can be evaluated and resolved along unfolded control flow paths using a SMT solver (YICES[5]) on the derived variable path predicates. We were able to demonstrate the potential of this approach in a case study of WS-BPEL programs supplied by an industrial partner [13]. Has there been no safe verification for these programs possible beforehand, based upon conventional program-to-Petri-net mappings, were we able to provide for precise low-level models and thus verification for half of the case study's programs using our system.

6 Related Work

In model checking, *predicate abstraction* [10] is used to exhaustively reason about infinitely many concrete program states in terms of finite number of abstract states as determined by a pre-defined set of predicates. *Counterexample-guided abstraction refinement* [4] in addition allows for iteratively refining predicates, such that a rather coarse abstraction is made more precise, based upon validating the abstract states, which have been identified as counterexamples by the model checking process, with respect to feasibility of their corresponding concrete states. Abstract states denoting false positives are thus identified and ruled out until an ideal abstraction is found. While generating predicates for describing program state in this way is accurate and precise, it is at the same time complex and expensive as it requires multiple iterations and powerful decision procedures. Furthermore, model checking in general is focussed on a specific program property, which determines the resulting abstraction and thus predicates.

[4] `http://service-technology.org/fiona/`
[5] `http://yices.csl.sri.com/`

A particular fraction of work considers the problem of infeasible paths as a cause of imprecision in data flow analysis. A common approach is to augment the analysis lattice with a set of fixed predicates or assertions on variable values, resulting in a so-called *qualified data flow problem* [14], which helps to avoid merging data flow values with contradicting assertions, e.g., infeasible paths. However, compared to the analysis described in this paper, the set of considered predicates is either limited to predicates appearing in branching conditions and which are only propagated as long as their value does not not change [2, 16] or pre-defined by a given set of predicates by means of a specification [7, 12]. The more advanced techniques in [8, 9] instead iteratively refine the set of considered predicates, for ruling out both, infeasible paths and imprecise merging of data flow facts, until a precise enough solution to the data flow problem is found.

Bodík et al. [2] apply demand-driven analysis for identifying infeasible paths by propagating branching predicates backwards until their value is determined. The used symbolic resolution mechanism is limited to constant assignments and condition predicates. In the same line of work falls [19], where predicates describing program state are derived in a backwards fashion based on the weakest precondition calculus. While the formal framework allows for arbitrary predicates, its implementation again confines considered predicates to simple variants $x\ op\ c$, where x is a variable and c a constant, to regain analysis effectiveness.

Similar methods have been used to identify false positives for static analysis using backwards symbolic execution [1, 11, 17]. These methods infer *path conditions*, i.e., predicates describing necessary requirements of program state for paths, which can be feed into constraint solvers. If insatisfiability is then shown, the paths and thus any associated program information are false positives.

To the knowledge of the authors, the presented analysis is the first data flow analysis for deriving predicates describing program state on a per variable basis. Furthermore, the authors are not aware of an analysis using the sparse analysis approach based upon SSA-form for deriving predicates describing program state.

7 Conclusion

In this paper, we have presented a novel data flow analysis based upon SSA-form for deriving variable predicates as predicate-based characterization of a program's variables' values. We have motivated how SSA-form benefits such an analysis by multiple means: First, exploiting the single assignment property allows us to use instruction and branching conditions as predicates for describing program state. Additionally, relevant parts of the program state can be easily identified for each variable following the def-use-chains implicitly given in SSA-form, which facilitates a sparse analysis. Eventually, Φ-functions depict variable definitions on confluent control flow paths and thus enable for the natural derivation of path information. While the variable predicates derived by our analysis have currently only been used for generating low-level models to more precisely model check WS-BPEL programs, we are confident on applying our analysis also to other programming languages and application domains in future work.

References

1. Arzt, S., Rasthofer, S., Hahn, R., Bodden, E.: Using Targeted Symbolic Execution for Reducing False-Positives in Dataflow Analysis. In: SOAP'15, Proc. pp. 1–6. ACM (2015)
2. Bodík, R., Gupta, R., Soffa, M.L.: Refining Data Flow Information Using Infeasible Paths. In: ESEC-FSE'97, Proc. pp. 361–377. ACM (1997)
3. Chow, F., Chan, S., Liu, S.M., Lo, R., Streich, M.: Effective Representation of Aliases and Indirect Memory Operations in SSA-Form. In: CC'05, Proc. pp. 253–267. Springer (2005)
4. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, M.: Counterexample-Guided Abstraction Refinement. In: CAV'00, Proc. pp. 154–169. Springer (2000)
5. Click, C., Cooper, K.D.: Combining Analyses, Combining Optimizations. ACM TOPLAS 17(2), 181–196 (1995)
6. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM TOPLAS 13(4), 451–490 (1991)
7. Das, M., Lerner, S., Seigle, M.: ESP: Path-Sensitive Program Verification in Polynomial Time. In: PLDI'02, Proc. pp. 57–68. ACM (2002)
8. Dhurjati, D., Das, M., Yang, Y.: Path-Sensitive Dataflow Analysis with Iterative Refinement. In: SAS'06, Proc. pp. 425–442. Springer (2006)
9. Fischer, J., Jhala, R., Majumdar, R.: Joining Dataflow with Predicates. In: ESEC-FSE'05, Proc. pp. 227–236. ACM (2005)
10. Graf, S., Saidi, H.: Construction of Abstract State Graphs with PVS. In: CAV'97, Proc. pp. 72–83. Springer (1997)
11. Hammer, C., Schaade, R., Snelting, G.: Static Path Conditions for Java. In: PLAS'08, Proc. pp. 57–66. ACM (2008)
12. Hampapuram, H., Yang, Y., Das, M.: Symbolic Path Simulation in Path-Sensitive Dataflow Analysis. In: PASTE'05, Proc. pp. 52–58. ACM (2005)
13. Heinze, T.S., Amme, W., Moser, S.: Compiling More Precise Petri Net Models for an Improved Verification of Service Implementations. In: SOCA 2014, Proc. pp. 25–32. IEEE (2014)
14. Holley, L.H., Rosen, B.K.: Qualified Data Flow Problems. In: POPL'80, Proc. pp. 68–82. ACM (1980)
15. Kam, J.B., Ullman, J.D.: Monotone Data Flow Analysis Frameworks. Acta Inf. 7(3), 305–317 (1977)
16. Murphy, B.R.: Frameworks for Precise Program Analysis. Ph.D. thesis, Stanford University (2001)
17. Snelting, G.: Combining Slicing and Constraint Solving for Validation of Measurement Software. In: SAS'96, Proc. pp. 332–348. Springer (1996)
18. Wegman, M.N., Zadeck, F.K.: Constant Propagation with Conditional Branches. ACM TOPLAS 13(2), 181–210 (1991)
19. Winter, K., Zhang, C., Hayes, I.J., Keynes, N., Cifuentes, C., Li, L.: Path-Sensitive Data Flow Analysis Simplified. In: ICFEM 2013, Proc. pp. 415–430. Springer (2013)
20. Web Services Business Process Execution Language Version 2.0. OASIS Standard (2007), `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`

A General Iterative Algorithm for Data Flow Analysis

A data flow framework is defined as tuple $(L, \wedge, F)$, which consists of the data flow information set L, the meet operator $\wedge$ on L, and the set F of semantic functions $f \in F$ with $f\colon L \to L$. Figure 4 shows the general iterative algorithm for data flow analysis, that terminates and yields the maximum fixpoint solution for a data flow framework iff the semantic functions are monotone and $(L, \wedge)$ forms a bounded semi lattice with a one element 1 and a zero element 0 [15].

Thereby, a semi lattice $(L, \wedge)$ is a set L with a binary operation $\wedge$ such that for all $a, b, c \in L$ holds $a \wedge a = a$ (idempotence), $a \wedge b = b \wedge a$ (commutativity), and $a \wedge (b \wedge c) = (a \wedge b) \wedge c$ (associativity). A semi lattice $(L, \wedge)$ has a zero element $0 \in L$ iff $a \wedge 0 = 0$ for every $a \in L$ and a one element $1 \in L$ iff $a \wedge 1 = a$ for every $a \in L$. A function $f\colon L \to L$ is called monotone iff for each $a, b \in L$ holds $a \leq b \to f(a) \leq f(b)$, whereby $\leq$ is the ordinary partial order induced by the semi lattice $(L, \wedge)$, i.e., $a \leq b \leftrightarrow a \wedge b = a$ for each $a, b \in L$.

In the initial phase of the algorithm, the data flow information that corresponds to the one element is assigned to each node other than the start node. The start node (s) gets a special element *NULL* assigned, depending on the considered data flow problem. In the iteration phase, the algorithm derives for each node successively the outgoing data flow information from its direct predecessor nodes. The algorithm terminates and yields a safe solution of the data flow framework, if for each node no further data flow information can be derived.

In general, the efficiency of the general iterative algorithm is influenced by the order in which nodes are visited. Reverse postorder is a typical iteration order for forward data flow problems. In contrast, postorder is often used for backward data flow problems. In addition, the general algorithm can be improved by noticing that the incoming data flow information will not change the outgoing information of a node. Therefore, data flow algorithms that are used in practice are often based on worklist approaches, in which nodes are kept that still need to be processed. In each iteration of such an algorithm, an arbitrary node is taken from the worklist, its outgoing information is computed, and if the outgoing information has been changed, the node's successors are added to the worklist.

B Analysis with Path Designators

The proof of correctness for the analysis with path designators resembles the correctness proof for the original analysis in Sect. 3. To this end, we first redefine the analysis in terms of a data flow framework MDF_{VP^*}:

Definition 7. *The data flow framework for the improved analysis with path designators is defined by the tuple* $MDF_{\mathrm{VP}^*} = (L_{\mathrm{VP}^*}, \wedge_{\mathrm{VP}^*}, F_{\mathrm{VP}^*})$ *such that* $L_{\mathrm{VP}^*} = \mathcal{P}(\mathit{Var} \times \mathcal{P}(N \times \mathbb{N}) \times \mathcal{P}(\mathit{Pred}))$ *and* $\wedge_{\mathrm{VP}^*} = \cup$. *Further, the semantic functions* F_{VP^*} *are defined according to instructions* $i \in \mathit{Instr}$ *as follows:*

- *Φ-function* $i\colon x = \Phi(x_1, \ldots, x_n)$
  $VP^*_{\mathrm{out}} = \mathit{update}^*_i(\mathit{remove}^*(VP^*_{\mathrm{in}}, V))$, *where*

```
INF_out(s) := NULL
foreach n ∈ N \ {s} do
   INF_out(n) := 1
end for
repeat
   stable := true
   foreach n ∈ N do
      INF_in(n):= ⋀_{n'∈predecessors(n)} INF_out(n')
      new := f_n(INF_in(n));
      if new ≠ INF_out(n) then
         INF_out(n) := new
         stable := false
      end if
   end for
until stable
```

Fig. 4. The general iterative algorithm for data flow analysis

$V \subseteq Var$ *is the set of all variables defined in* $node(x)$

- *any other instruction* i *defining value* x
  $VP^*_{\text{out}} = update^*_i(remove^*(VP^*_{\text{in}}, \{x\}))$

where $remove^* \colon L_{\text{VP}^*} \times \mathcal{P}(Var) \to L_{\text{VP}^*}$ *is defined for* $k \in L_{\text{VP}^*}, V \subseteq Var$ *by:*

- $remove^*(k, V) = \{(y, \delta, p \cap \bigcap_{v \in V} \overline{Pred_v}) \mid (y, \delta, p) \in k \wedge y \notin V\}$

and $update^*_i \colon L_{\text{VP}^*} \to L_{\text{VP}^*}$ *is defined for* $l \in L_{\text{VP}^*}$ *according to* $i \in Instr$ *by:*

- *constant assignment* $i \colon x = c$
  $update^*_i(l) = l \cup \{(x, \emptyset, \{x = c\})\}$
- *simple assignment* $i \colon x = y$
  $update^*_i(l) = l \cup \{(x, \delta, p \cup \{x = y\}) \mid (y, \delta, p) \in l\}$
- *complex assignment* $i \colon x = y \ op \ z$
  $update^*_i(l) = l \cup \{(x, \delta \cup \gamma, p \cup q \cup \{x = y \ op \ z\}) \mid (y, \delta, p), (z, \delta, q) \in l \wedge (\delta \cup \gamma) \ is \ definite\}$
- *assertion* $i \colon x = assert(y \ op \ c)$
  $update^*_i(l) = l \cup \{(x, \delta, p \cup \{x = y, x \ op \ c\}) \mid (y, \delta, p) \in l\}$
- *Φ-function* $i \colon x = \Phi(x_1, \ldots, x_n)$
  $update^*_i(l) = l \cup \bigcup_{1 \le j \le n} \{(x, \delta \oplus \{(node(x), j)\}, p \cup \{x = x_j\}) \mid (x_j, \delta, p) \in l\}$

With Definition 7 in place, we then show in the following two lemmata that MDF_{VP^*} is a monotone data flow framework:

Lemma 3. $(L_{\text{VP}^*}, \wedge_{\text{VP}^*})$ *is a bounded semi lattice with zero element* $0 \in L_{\text{VP}^*}$ *and one element* $1 \in L_{\text{VP}^*}$ *such that* $\forall l \in L_{\text{VP}^*} \colon \ l \wedge_{\text{VP}^*} 1 = l$ *and* $l \wedge_{\text{VP}^*} 0 = 0$.

Proof. For a given program, sets *Var*, N, and *Instr* are finite. Furthermore, the number of a join node's predecessors is limited by a number $n \geq 0$, so that, by construction of the semantic functions, the analysis lattice effectively becomes $L_{\mathrm{VP}^*} = \mathcal{P}(\mathit{Var} \times \mathcal{P}(N \times \{0, 1, \ldots, n\}) \times \mathcal{P}(\mathit{Pred}))$, and is also finite. The lemma follows from the fact that for a finite set M, $(\mathcal{P}(M), \cup)$ is a bounded semi lattice with zero element M and one element $\emptyset$. □

Lemma 4. *The semantic functions F_{VP^*} are monotone.*

Proof. Similar to Lemma 2, we prove for each semantic function $f \in F_{\mathrm{VP}^*}$ that $f(l) \cup f(k) \subseteq f(l \sqcup k)$ holds. Therefore, we show for $l, k \in L_{\mathrm{VP}^*}$ and $V \subseteq \mathit{Var}$ that $\mathit{remove}^*(l, V) \cup \mathit{remove}^*(k, V) = \mathit{remove}^*(l \sqcup k, V)$:

- $\mathit{remove}^*(l, V) \cup \mathit{remove}^*(k, V)$
  $= \{(y, \delta, p \cap \bigcap_{v \in V} \overline{\mathit{Pred}_v} \mid (y, \delta, p) \in l \wedge y \notin V\}$
  $\quad \cup \{(y, \delta, p \cap \bigcap_{v \in V} \overline{\mathit{Pred}_v} \mid (y, \delta, p) \in k \wedge y \notin V\}$
  $= \{(y, \delta, p \cap \bigcap_{v \in V} \overline{\mathit{Pred}_v} \mid (y, \delta, p) \in l \cup k \wedge y \notin V\}$
  $= \mathit{remove}^*(l \sqcup k, V)$

and that $\mathit{update}_i^*(l) \cup \mathit{update}_i^*(k) \subseteq \mathit{update}_i^*(l \sqcup k)$ for every $i \in \mathit{Instr}$:

- constant assignment $i\colon x = c$
  $\mathit{update}_i^*(l) \cup \mathit{update}_i^*(k)$
  $= l \cup \{(x, \emptyset, \{x = c\})\} \cup k \cup \{(x, \emptyset, \{x = c\})\}$
  $= l \cup k \cup \{(x, \emptyset, \{x = c\})\}$
  $= \mathit{update}_i^*(l \sqcup k)$
- simple assignment $i\colon x = y$
  $\mathit{update}_i^*(l) \cup \mathit{update}_i^*(k)$
  $= l \cup \{(x, \delta, p \cup \{x = y\}) \mid (y, \delta, p) \in l\} \cup k \cup \{(x, \delta, p \cup \{x = y\}) \mid (y, \delta, p) \in k\}$
  $= l \cup k \cup \{(x, \delta, p \cup \{x = y\}) \mid (y, \delta, p) \in l \cup k\}$
  $= \mathit{update}_i^*(l \sqcup k)$
- Φ-function $i\colon x = \Phi(x_1, \ldots, x_n)$
  $\mathit{update}_i^*(l) \cup \mathit{update}_i^*(k)$
  $= l \cup \bigcup_{1 \leq j \leq n} \{(x, \delta \oplus \{(\mathit{node}(x), j)\}, p \cup \{x = x_j\}) \mid (x_j, \delta, p) \in l\}$
  $\quad \cup k \cup \bigcup_{1 \leq j \leq n} \{(x, \delta \oplus \{(\mathit{node}(x), j)\}, p \cup \{x = x_j\}) \mid (x_j, \delta, p) \in k\}$
  $= l \cup k \cup \bigcup_{1 \leq j \leq n} \{(x, \delta \oplus \{(\mathit{node}(x), j)\}, p \cup \{x = x_j\}) \mid (x_j, \delta, p) \in l \cup k\}$
  $= \mathit{update}_i^*(l \sqcup k)$

– complex assignment $i\colon x = y\ op\ z$

$$
\begin{aligned}
&update_i^*(l) \cup update_i^*(k)\\
&= l \cup \{(x, \delta \cup \gamma, p \cup q \cup \{x = y\ op\ z\}) \mid (y,\delta,p),(z,\gamma,q) \in l \wedge (\delta \cup \gamma) \text{ is definite}\}\\
&\quad \cup k \cup \{(x, \delta \cup \gamma, p \cup q \cup \{x = y\ op\ z\}) \mid (y,\delta,p),(z,\gamma,q) \in k \wedge (\delta \cup \gamma) \text{ is definite}\}\\
&\subseteq l \cup k\\
&\quad \cup \{(x, \delta \cup \gamma, p \cup q \cup \{x = y\ op\ z\}) \mid (y,\delta,p),(z,\gamma,q) \in l \cup k \wedge (\delta \cup \gamma) \text{ is definite}\}\\
&= update_i^*(l \cup k)
\end{aligned}
$$

– assertion $i\colon x = assert(y\ op\ c)$

$$
\begin{aligned}
&update_i^*(l) \cup update_i^*(k)\\
&= l \cup \{(x, \delta, p \cup \{x = y, x\ op\ c\}) \mid (y,\delta,p) \in l\}\\
&\quad \cup k \cup \{(x, \delta, p \cup \{x = y, x\ op\ c\}) \mid (y,\delta,p) \in k\}\\
&= l \cup k \cup \{(x, \delta, p \cup \{x = y, x\ op\ c\}) \mid (y,\delta,p) \in l \cup k\}\\
&= update_i^*(l \cup k)
\end{aligned}
$$

From the fact that the composition of monotone functions $remove^*$ and $update_i^*$ is also a monotone function, then again follows the lemma. □

Theorem 2. *The general iterative algorithm terminates with the maximum fixpoint solution for each instance of the data flow framework* MDF_{VP^*}.

Proof. This immediately follows from MDF_{VP^*} being a monotone data flow framework according to Lemma 3 and Lemma 4. □

C Analysis with Unreachable Code Elimination

The proof of correctness for the analysis with unreachable code elimination again resembles the correctness proof in Sect. 3. To this end, we redefine the combined analysis in terms of a data flow framework $MDF_{\mathrm{VP}^\dagger}$. Following [5], the data flow framework is based on the product of the original lattice $(L_{\mathrm{VP}}, \wedge_{\mathrm{VP}})$ and a boolean lattice with elements $\mathcal{U}$ and $\mathcal{R}$ modelling reachability information:

Definition 8. *The data flow framework for the combined analysis with unreachable code elimination is defined by the tuple* $MDF_{\mathrm{VP}^\dagger} = (L_{\mathrm{VP}^\dagger}, \wedge_{\mathrm{VP}^\dagger}, F_{\mathrm{VP}^\dagger})$ *such that* $L_{\mathrm{VP}^\dagger} = \{(\mathcal{U},\emptyset)\} \cup \{(\mathcal{R}, r) \mid r \in L_{\mathrm{VP}}\} \subseteq \{\mathcal{U},\mathcal{R}\} \times L_{\mathrm{VP}}$ *and*

$$
k \wedge_{\mathrm{VP}^\dagger} l = \begin{cases} (\mathcal{U},\emptyset) & \text{if } l = (\mathcal{U},\emptyset) \text{ and } k = (\mathcal{U},\emptyset)\\ (\mathcal{R},s) & \text{if } l = (\mathcal{R},s) \text{ and } k = (\mathcal{U},\emptyset)\\ (\mathcal{R},t) & \text{if } l = (\mathcal{U},\emptyset) \text{ and } k = (\mathcal{R},t)\\ (\mathcal{R},s \cup t) & \text{if } l = (\mathcal{R},s) \text{ and } k = (\mathcal{R},t) \end{cases}
$$

for every $k, l \in_{\mathrm{VP}^\dagger}$ *and* $s, t \in L_{\mathrm{VP}}$. *Furthermore, the semantic functions* $F_{\mathrm{VP}^\dagger}$ *are defined according to instructions* $i \in Instr$ *as follows:*

– *branch guard instruction* $i\colon guard(x\ op\ c)$

$$
VP_{\mathrm{out}}^\dagger = \begin{cases} (\mathcal{R},s) & \text{if } VP_{\mathrm{in}}^\dagger = (\mathcal{R},s) \text{ and } h \models \neg(x\ op\ c) \text{ can}\\ & \quad \text{not be shown for } h = \{p \mid (x,p) \in s\}\\ (\mathcal{U},\emptyset) & \text{otherwise} \end{cases}
$$

– *Φ-function* i: $x = \Phi(x_1, \ldots, x_n)$

$$VP^{\dagger}_{\text{out}} = \begin{cases} (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, V))) & \text{if } VP^{\dagger}_{\text{in}} = (\mathcal{R}, s) \\ (\mathcal{U}, \emptyset) & \text{otherwise} \end{cases},$$

where $V \subseteq \mathit{Var}$ *is the set of all variables defined in* $\mathit{node}(x)$

– *any other instruction* i *defining value* x

$$VP^{\dagger}_{\text{out}} = \begin{cases} (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, \{x\}))) & \text{if } VP^{\dagger}_{\text{in}} = (\mathcal{R}, s) \\ (\mathcal{U}, \emptyset) & \text{otherwise} \end{cases}$$

where L_{VP}*,* remove*, and* update_i *are defined as in Sect. 3.*

With Definition 8 in place, we then show in the following two lemmata that $MDF_{\text{VP}^\dagger}$ is a monotone data flow framework:

Lemma 5. $(L_{\text{VP}^\dagger}, \wedge_{\text{VP}^\dagger})$ *is a bounded semi lattice with zero element* $0 \in L_{\text{VP}^\dagger}$ *and one element* $1 \in L_{\text{VP}^\dagger}$ *such that* $\forall l \in L_{\text{VP}^\dagger} \colon\ l \wedge_{\text{VP}^\dagger} 1 = l$ *and* $l \wedge_{\text{VP}^\dagger} 0 = 0$.

Proof. Tuple $(\mathbb{B}, \vee)$ is obviously a bounded semi lattice with zero element *true* and one element *false*. Lemma 1 also shows that $(L_{\text{VP}}, \wedge_{\text{VP}})$ is a bounded semi lattice with zero element $\mathit{Var} \times \mathcal{P}(\mathit{Pred})$ and one element $\emptyset$. Considering $\mathcal{R} = \mathit{true}$ and $\mathcal{U} = \mathit{false}$, $(L_{\text{VP}^\dagger}, \wedge_{\text{VP}^\dagger})$ can be seen as cartesian product of bounded semi lattices, which forms again a bounded semi lattice. Further, for all $l \in L_{\text{VP}^\dagger}$, we have $l \wedge^{\dagger}_{\text{VP}} 0 = 0$ and $l \wedge_{\text{VP}^\dagger} 1 = l$ for $0 = (\mathcal{R}, \mathit{Var} \times \mathcal{P}(\mathit{Pred}))$ and $1 = (\mathcal{U}, \emptyset)$ □

Lemma 6. *The semantic functions* $F_{\text{VP}^\dagger}$ *are monotone.*

Proof. We first consider semantic functions $f \in F_{\text{VP}^\dagger}$ for branch guard instructions. Using $l \leq_{\text{VP}^\dagger} k \leftrightarrow l \wedge_{\text{VP}^\dagger} k = l$, we will show that $l \leq_{\text{VP}^\dagger} k$ implies $f(l) \leq_{\text{VP}^\dagger} f(k)$ for each $l, k \in L_{\text{VP}^\dagger}$:

– branch guard instruction i: $\mathit{guard}(x\ \mathit{op}\ c)$

We consider cases for l, k

- $l = (\mathcal{U}, \emptyset)$, $k = (\mathcal{U}, \emptyset)$: $f(l) = (\mathcal{U}, \emptyset) = f(k)$
- $l = (\mathcal{U}, \emptyset)$, $k = (\mathcal{R}, s)$ with $s \in L_{\text{VP}}$: contradicts premise $l \leq_{\text{VP}^\dagger} k$
- $l = (\mathcal{R}, s)$, $k = (\mathcal{U}, \emptyset)$ with $s \in L_{\text{VP}}$:
  $f(l) \leq_{\text{VP}^\dagger} (\mathcal{U}, \emptyset) = f(k)$ since $(\mathcal{U}, \emptyset)$ is one element of $(L_{\text{VP}^\dagger}, \wedge_{\text{VP}^\dagger})$
- $l = (\mathcal{R}, s)$, $k = (\mathcal{R}, t)$ with $s, t \in L_{\text{VP}}$:
  We consider cases for $g = \{p \mid (x, p) \in s\}$, $h = \{p \mid (x, p) \in t\}$:
  * $g \models \neg(x\ \mathit{op}\ c)$ can not be shown, $h \models \neg(x\ \mathit{op}\ c)$ can not be shown:
    $f(l) = (\mathcal{R}, s) \leq_{\text{VP}^\dagger} (\mathcal{R}, t) = f(k)$ due to premise $l \leq_{\text{VP}^\dagger} k$
  * $g \models \neg(x\ \mathit{op}\ c)$, $h \models \neg(x\ \mathit{op}\ c)$ can not be shown:
    contradicts premise $l \leq_{\text{VP}^\dagger} k$, i.e., from $l = (\mathcal{R}, s) \leq_{\text{VP}^\dagger} (\mathcal{R}, t) = k$ we have $t \subseteq s$ and $h = \{p \mid (x, p) \in t\} \subseteq \{p \mid (x, p) \in s\} = g$, assuming $g \models \neg(x\ \mathit{op}\ c)$ we get $h \models \neg(x\ \mathit{op}\ c)$ violating the case condition

* $g \models \neg(x\ op\ c)$ can not be shown, $h \models \neg(x\ op\ c)$:
  $f(l) = (\mathcal{R}, s) \leq_{\mathrm{VP}^\dagger} (\mathcal{U}, \emptyset) = f(k)$
* $g \models \neg(x\ op\ c)$, $h \models \neg(x\ op\ c)$:
  $f(l) = (\mathcal{U}, \emptyset) = f(k)$

Similar to Lemma 2, we show $f(l \wedge_{\mathrm{VP}^\dagger} k) \leq_{\mathrm{VP}^\dagger} f(l) \wedge_{\mathrm{VP}^\dagger} f(k)$ for the remaining semantic functions $f \in F_{\mathrm{VP}^\dagger}$, with $V \subseteq \mathit{Var}$ and $l, k \in L_{\mathrm{VP}^\dagger}$:

– Φ-function i: $x = \Phi(x_1, \ldots, x_n)$

  We consider cases for l, k

  - $l = (\mathcal{U}, \emptyset)$, $k = (\mathcal{U}, \emptyset)$:
    $f(l) \wedge_{\mathrm{VP}^\dagger} f(k) = (\mathcal{U}, \emptyset) \wedge_{\mathrm{VP}^\dagger} (\mathcal{U}, \emptyset) = (\mathcal{U}, \emptyset) = f((\mathcal{U}, \emptyset)) = f(l \wedge_{\mathrm{VP}^\dagger} k)$
  - $l = (\mathcal{R}, s)$, $k = (\mathcal{U}, \emptyset)$ with $s \in L_{\mathrm{VP}}$:
    $f(l) \wedge_{\mathrm{VP}^\dagger} f(k) = (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, V)) \wedge_{\mathrm{VP}^\dagger} (\mathcal{U}, \emptyset)$
    $= (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, V)) = f((\mathcal{R}, s)) = f(l \wedge_{\mathrm{VP}^\dagger} k)$
  - $l = (\mathcal{U}, \emptyset)$, $k = (\mathcal{R}, s)$ with $s \in L_{\mathrm{VP}}$:
    $f(l) \wedge_{\mathrm{VP}^\dagger} f(k) = (\mathcal{U}, \emptyset) \wedge_{\mathrm{VP}^\dagger} (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, V))$
    $= (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, V)) = f((\mathcal{R}, s)) = f(l \wedge_{\mathrm{VP}^\dagger} k)$
  - $l = (\mathcal{R}, s)$, $k = (\mathcal{R}, t)$ with $s, t \in L_{\mathrm{VP}}$:
    $f(l) \wedge_{\mathrm{VP}^\dagger} f(k)$
    $= (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, V)) \wedge_{\mathrm{VP}^\dagger} (\mathcal{R}, \mathit{update}_i(\mathit{remove}(t, V))$
    $= (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, V) \cup \mathit{update}_i(\mathit{remove}(t, V))$
    $= (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s \cup t, V)) = f((\mathcal{R}, s \cup t)) = f(l \wedge_{\mathrm{VP}^\dagger} k)$
    since $\mathit{update}_i(\mathit{remove}(s, \{x\}) \cup \mathit{update}_i(\mathit{remove}(t, \{x\})$
    $= \mathit{update}_i(\mathit{remove}(s \cup t, \{x\})$ according to Lemma 2

– any other instruction i defining value x

  We consider cases for l, k

  - $l = (\mathcal{U}, \emptyset)$, $k = (\mathcal{U}, \emptyset)$:
    $f(l) \wedge_{\mathrm{VP}^\dagger} f(k) = (\mathcal{U}, \emptyset) \wedge_{\mathrm{VP}^\dagger} (\mathcal{U}, \emptyset) = (\mathcal{U}, \emptyset) = f((\mathcal{U}, \emptyset)) = f(l \wedge_{\mathrm{VP}^\dagger} k)$
  - $l = (\mathcal{R}, s)$, $k = (\mathcal{U}, \emptyset)$ with $s \in L_{\mathrm{VP}}$:
    $f(l) \wedge_{\mathrm{VP}^\dagger} f(k) = (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, \{x\})) \wedge_{\mathrm{VP}^\dagger} (\mathcal{U}, \emptyset)$
    $= (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, \{x\})) = f((\mathcal{R}, s)) = f(l \wedge_{\mathrm{VP}^\dagger} k)$
  - $l = (\mathcal{U}, \emptyset)$, $k = (\mathcal{R}, s)$ with $s \in L_{\mathrm{VP}}$:
    $f(l) \wedge_{\mathrm{VP}^\dagger} f(k) = (\mathcal{U}, \emptyset) \wedge_{\mathrm{VP}^\dagger} (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, \{x\}))$
    $= (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, \{x\})) = f((\mathcal{R}, s)) = f(l \wedge_{\mathrm{VP}^\dagger} k)$
  - $l = (\mathcal{R}, s)$, $k = (\mathcal{R}, t)$ with $s, t \in L_{\mathrm{VP}}$:
    $f(l) \wedge_{\mathrm{VP}^\dagger} f(k)$
    $= (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, \{x\})) \wedge_{\mathrm{VP}^\dagger} (\mathcal{R}, \mathit{update}_i(\mathit{remove}(t, \{x\}))$
    $= (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s, \{x\}) \cup \mathit{update}_i(\mathit{remove}(t, \{x\}))$
    $\geq_{\mathrm{VP}^\dagger} (\mathcal{R}, \mathit{update}_i(\mathit{remove}(s \cup t, \{x\})) = f((\mathcal{R}, s \cup t)) = f(l \wedge_{\mathrm{VP}^\dagger} k)$
    since $\mathit{update}_i(\mathit{remove}(s, \{x\}) \cup \mathit{update}_i(\mathit{remove}(t, \{x\})$
    $\subseteq \mathit{update}_i(\mathit{remove}(s \cup t, \{x\})$ according to Lemma 2 $\square$

Theorem 3. *The general iterative algorithm terminates with the maximum fixpoint solution for each instance of the data flow framework* $MDF_{\mathrm{VP}^\dagger}$.

Proof. This immediately follows from MDF_{VP^*} being a monotone data flow framework according to Lemma 5 and Lemma 6. □

Following the same line of arguments, we can as well combine the improved analysis with path designators with unreachable code elimination:

Corollary 4. *The general iterative algorithm terminates with the maximum fixpoint solution for each instance of a data flow framework representing a combination of the analysis with path designators and unreachable code elimination.*