Software Product Line Analysis Using Variability-aware Datalog

Ramy Shahin*, Murad Akhundov*, Marsha Chechik*

*University of Toronto, Canada

Abstract—Applying program analyses to Software Product Lines (SPLs) has been a fundamental research problem at the intersection of Product Line Engineering and software analysis. Different attempts have been made to “lift” particular product-level analyses to run on the entire product line. In this paper, we tackle the class of Datalog-based analyses (e.g., pointer and taint analyses), study the theoretical aspects of lifting Datalog inference, and implement a lifted inference algorithm inside the Soufflé Datalog engine. We evaluate our implementation on a set of Java and C-language benchmark product lines. We show significant savings in processing time and fact database size (billions of times faster on one of the benchmarks) compared to brute-force analysis of each product individually.

Index Terms—Software Product Lines, Datalog, Program Analysis, Pointer Analysis, Lifting, Variability, Doop, Soufflé.

1 INTRODUCTION

Software Product Lines (SPLs) are families of related products, usually developed together from a common set of artifacts. Each product configuration is a combination of features. As a result, the number of potential products is combinatorial in the number of features. This high level of configurability is usually desired. However, analysis tools (syntax analyzers, type checkers, model checkers, static analysis tools, etc...) typically work on a single product, not the whole SPL. Applying an analysis to each product separately is usually infeasible for non-trivial SPLs because of the exponential number of products [1].

Since all products of an SPL share a common set of artifacts, analyzing each product individually (usually referred to as *brute-force analysis*) would involve a lot of redundancy. How to leverage this commonality and analyze the whole product line at once, bringing the total analysis time down, is a fundamental research problem at the intersection of Product Line Engineering and software analysis. Different attempts have been made to *lift* individual analyses to run on product lines [2]–[8]. Those attempts show significant time savings when the SPL is analyzed as a whole compared to brute-force analysis. The downside though is the amount of effort required to correctly lift each of those analyses.

In this paper, we tackle the class of Datalog-based program analyses. Datalog is a declarative query language that adds logical inference to relational queries. Some program analyses (in particular, pointer and taint analyses) can be fully specified as sets of Datalog inference rules. Those rules are applied by an inference engine to facts extracted from a software product. Results are more facts, inferred by the engine based on the rules. The advantage of Datalog-based analyses is that they are declarative, concise and can be efficiently executed by highly optimized Datalog engines [9], [10].

Instead of lifting individual Datalog-based analyses, we lift a Datalog engine. This way any analysis running on the

lifted engine is lifted for free. Our approach is not specific to a particular engine though, and can be implemented in others.

Contributions In this paper we make the following contributions: (1) We present *infer*, a Datalog inference algorithm lifted to facts extracted from Software Product Lines. (2) We state the correctness criteria of lifted Datalog inference and show that *infer* is correct. (3) We implement our lifted algorithm as a part of a Datalog engine. We also extend the Doop pointer analysis framework [11] to extract facts from SPLs. (4) We evaluate our implementation on a sample of pointer and taint analyses applied to a suite of Java benchmarks, and a set of hand-crafted analyses of C programs applied to the BusyBox product line. We show significant savings in processing time and fact database sizes compared to brute-force analysis of one product at a time. For one of the benchmarks, our lifted implementation is billions of times faster than brute-force analysis (with savings in database size of the same order of magnitude).

In this paper we are extending and revising an ESEC/FSE conference paper [12]. In particular: (1) We are evaluating our approach on a wider range of product line benchmarks written in Java and C-language, and using different feature annotation mechanisms; (2) The distinction between annotation mechanisms (C PreProcessor (CPP) and CIDE) is discussed and illustrated, with emphasis on its impact on variability-aware fact extraction; (3) Variability-aware fact extraction is discussed in more detail; (4) We added more elaborate examples and illustrations throughout the paper.

The rest of the paper starts with a background on SPLs and Datalog (Sec. 2). We provide a theoretical treatment of Datalog inference, how the inference algorithm is lifted, together with correctness criteria and a correctness proof in Sec. 3. In Sec. 4, we describe the implementation of our algorithm in the Soufflé engine. Evaluation process and results are discussed in Sec. 5. We compare our approach

to related work in Sec. 6 and conclude (Sec. 7).

2 BACKGROUND

In this section, we summarize the basic concepts of Software Product Lines, Horn Clauses, Datalog and Datalog-based analyses.

2.1 Software Product Lines

A *Software Product Line (SPL)* is a family of related software products developed together. Different variants of an SPL have different *features*, i.e., externally visible attributes such as a piece of functionality, support for a particular peripheral device, or a performance optimization.

Definition 1 (SPL). An SPL $\mathcal{L}$ is a tuple (F, Φ, D, ϕ) where: (1) F is the set of features s.t. an individual product can be derived from $\mathcal{L}$ via a *feature configuration* $\rho \subseteq F$. (2) $\Phi \in \text{Prop}(F)$ is a propositional formula over F defining the valid set of feature configurations. Φ is called a *Feature Model (FM)*. The set of valid configurations defined by Φ is called $\text{Conf}(\mathcal{L})$. (3) D is a set of program elements, called the *domain model*. The whole set of program elements is sometimes referred to as the *150% representation*. (4) $\phi : D \rightarrow \text{Prop}(F)$ is a total function mapping each program element to a proposition (*feature expression*) defined over the set of features F . $\phi(e)$ is called the *Presence Condition (PC)* of element e , i.e. the set of product configurations in which e is present.

There are different techniques for implementing feature variability in product lines. In *annotative* SPLs, the code of a single feature can be scattered across the product line code base, and code snippets belonging to that feature are explicitly annotated by some feature identifier. *Compositional* SPLs on the other hand modularize each feature as a separate unit, and then features are composed together based on the desired product configuration. Examples of compositional techniques are Aspect-Oriented Programming [13] and Feature-Oriented Programming [14]. *Delta-oriented* SPLs [15] are a third category of product lines, where a base implementation common across product variants is provided, and then feature-specific *deltas* with respect to that base implementation are provided. In this paper, we only focus on annotative SPL mechanisms.

We consider two variability-annotation mechanisms in this paper: C Pre-Processor (CPP) macro annotations, and CIDE [16] color-based annotations of source code.

CPP Example. The *annotative* Java product line in Listing 1 has the feature set $F = \{FA, FB\}$. Features are annotated using the *C Pre-Processor(CPP)* conditional compilation directives. By defining or not-defining macros corresponding to features, different products can be generated from this product line. One example is the product on Listing 2, with FA not defined and FB defined.

Here a single code-base (domain model D) is maintained, where different pieces of code are *annotated* with feature expressions. For example, the syntactic tokens on line 10 are annotated with $\neg FA$. That is, $\neg FA$ is the Presence Condition (PC) of these tokens. Similarly, tokens on line 13

```

1      class Parent {public Object f;}
2      class ClassA extends Parent {}
3      class ClassB extends Parent {}
4
5      ClassA o1 = new ClassA();
6      ClassB o2 = new ClassB();
7      #ifdef FA
8      Parent o3 = o1;
9      #else
10     Parent o3 = o2;
11     #endif
12     #ifdef FB
13     o2.f = o1;
14     #else
15     o2.f = o2;
16     #endif
17     Object r = o3.f;
```

Listing 1: A product line with features FA and FB .

```

1      Class Parent {public Object f;}
2      Class ClassA extends Parent {}
3      Class ClassB extends Parent {}
4
5      ClassA o1 = new ClassA();
6      ClassB o2 = new ClassB();
7      Parent o3 = o2;
8      o2.f = o1;
9      Object r = o3.f;
```

Listing 2: A product with a configuration $(\neg FA \wedge FB)$.

have the PC FB . This SPL allows all four feature combinations, so its feature model Φ is the propositional formula True .

CIDE Example. The code snippet in Fig. 1 comes from the Graph Product Line (GPL) benchmark [17], with feature-specific code highlighted in different colors. In this example, the light-blue color denotes the *EdgeObjects* feature, while the violet color denotes the feature combination $\text{EdgeObjects} \wedge \text{Weighted}$.

The CIDE annotation system enforces syntactic correctness and type-checking of annotated code. For example, because the declaration of the `weight` field is highlighted in violet, all code snippets referring to this field are highlighted in the same color.

The Feature Model (FM) of a product line can be represented graphically in a *Feature Diagram (FD)*. Fig. 2 is the FD of the GPL product line. Features (abstract and concrete) are recursively decomposed in tree-form, where all leaves are concrete features. Some features are marked as mandatory (they must exist in each product instance) and others as optional. Mutually-exclusive features are marked using the *alternative* notation. Inter-tree dependencies between features can be added on top of the tree structure of the FD using explicit propositional invariants (the propositional equalities below the diagram).

```

5 public class Edge implements EdgeIfc {
6 private Vertex start;
7 private Vertex end;
8 private int weight;
9
10 public Edge(Vertex the_start, Vertex the_end,
11             int aweight) {
12     } {
13     start = the_start;
14     end = the_end;
15     weight = aweight;
16 }
17
18 public void adjustAdorns(EdgeIfc the_edge) {
19     setWeight(the_edge.getWeight());
20 }
21
22 public void setWeight(int weight) {
23     this.weight = weight;
24 }
25
26 public int getWeight() {
27     return this.weight;
28 }
29
30 public Vertex getOtherVertex(Vertex vertex) {
31     if (vertex == start)
32         return end;
33     else if (vertex == end)
34         return start;
35     else
36         return null;
37 }

```

Figure 1: Example of a CIDE annotated code snippet.

2.2 Horn Clauses and Datalog

2.2.1 Horn Clauses

A *Horn Clause (HC)* is a disjunction of unique propositional literals with at most one positive literal. For example, $(\neg a \vee \neg b \vee c \vee \neg d)$ is an HC, which can be also written as a reverse-implication $(c \leftarrow (a \wedge b \wedge d))$, where c is called the *head* and $(a \wedge b \wedge d)$ is called the *body* of the clause. The language of HCs is a fragment of Propositional Logic that can be checked for satisfiability in linear time, as opposed to general propositional satisfiability which is NP-complete [18].

2.2.2 Datalog

Datalog is a declarative database query language that extends relational algebra with logical inference [19]. Datalog inference *rules* are HCs in First Order Logic, where atoms are predicate expressions, not just propositional literals. A *fact* is a ground rule with only a head and no body. Syntactically, the $\leftarrow$ symbol is usually used instead of backward implication, and atoms in the body are separated by commas instead of the conjunction symbol.

For example, given a binary predicate *Parent*, a binary predicate *Ancestor* can be defined using the following two rules:

```

Ancestor(x, y) :- Parent(x, y).
Ancestor(x, z) :- Parent(x, y), Ancestor(y, z).

```

Input: IDB, EDB

Result: EDB $\cup$ {inferred clauses}

```

1 repeat
2     fixpoint  $\leftarrow$  True;
3     foreach ( $C \leftarrow s_1, \dots, s_n$ )  $\in$  IDB do
4         foreach ( $\gamma, f_1, \dots, f_n, f_i \in$  EDB,  $[\gamma]s_i = f_i$  do
5             if  $[\gamma]C \notin$  EDB then
6                 fixpoint  $\leftarrow$  False;
7                 EDB  $\leftarrow$  EDB  $\cup$   $\{[\gamma]C\}$ ;
8             end
9         end
10    end
11 until fixpoint;
12 return EDB;

```

Algorithm 1: Inference algorithm infer (forward chaining).

That is, if x is a parent of y , then x is an ancestor of y . In addition, if x is a parent of y , and y is an ancestor of z , then x is an ancestor of z .

Fig. 3a defines the grammar of Datalog clauses as follows: (1) building blocks are finite sets of constants, variables and predicate symbols; (2) a *term* is a constant or a variable symbol; (3) a predicate expression is an n -ary predicate applied to arguments; (4) a *fact* is a ground predicate expression, i.e., all of its arguments are constants; (5) a *rule* is a Horn Clause of predicate expressions; and (6) a *Datalog clause* is either a fact or a rule.

A Datalog program is a finite set of rules, usually referred to as the *Intensional Database (IDB)*, which operates on a finite set of facts called the *Extensional Database (EDB)*. The inference algorithm (explained next) repeatedly applies the rules to the facts, inferring new facts and adding them to the EDB, until a fixed point is reached (i.e., no more new facts can be inferred).

2.2.3 Inference Algorithm (Algorithm 1)

For each rule R , the algorithm checks to see if the EDB has facts fulfilling the premises of R , with a consistent assignment of variables to constants (Fig. 3b). If it does, the head of that rule is inferred as a new fact F . If F doesn't already exist in the EDB, it is added to it. Newly inferred facts may trigger some of the rules again; this process continues until a fixed point is reached, i.e., no new facts are inferred. This algorithm (called *forward chaining* [20]) is guaranteed to terminate because it does not create any new constants, and runs in polynomial time w.r.t. the number of input clauses [20].

2.3 Example of a Datalog Analysis

Some program analyses [11], [21]–[23] can be written in Datalog as sets of clauses. Facts relevant to the analysis are extracted from the program to be analyzed, and then fed into a Datalog engine together with the analysis clauses. Fact extraction is usually analysis-specific because different analyses work on different aspects of the program. One example of Datalog-based analyses is pointer analysis.

Figure 2: Feature diagram of the GPL product line.

S = {finite set of constant symbols}
 V = {finite set of variables}
 P = {finite set of predicate symbols}
 $T ::= S \mid V$
 $L ::= P(T_1, \dots, T_n)$
 $F ::= P(S_1, \dots, S_m)$
 $R ::= L_0 :- L_1, \dots, L_k$
 $D ::= F \mid R$

(a) Datalog Grammar.

$\gamma : V \rightarrow S$
 $[\gamma]C = C[v/\gamma(v)]$, for each free variable v in C
 (b) Variable assignment function and substitution for clause C .

$PC ::= f \mid \neg PC \mid PC \wedge PC \mid PC \vee PC$
 $\hat{D} ::= (F, PC) \mid R$

(c) Grammar for lifted Datalog clauses. Syntactic category f is set of feature names.

Figure 3: (a) Grammar of Datalog clauses, (b) variable assignment function and substitution, and (c) lifted Datalog clauses.

Pointer analysis [24] determines which objects might be pointed to by a particular program expression. This whole-program analysis is *over-approximating* in the sense that it returns a set of objects that *might* be pointed to by each pointer, possibly with false positives. Fig. 4a shows a set of Datalog rules for a simple pointer analysis [25]. Each predicate defines a relation between different artifacts. For example, `VarPointsTo(v, h)` states that pointer v might point to heap object h . The first three rules specify the conditions for this predicate to hold: either a new object is allocated and a pointer is initialized; a pointer that already points to an object is assigned to another pointer; or an object field points to a heap object, and that field is assigned to another pointer. The fourth rule states that assigning a value to an object field results in that field pointing to the same object as the right-hand-side of the assignment.

Fig. 4b shows the facts corresponding to the program in Listing 2. The first two are object allocation facts; the third is an assignment fact, and the fourth and the fifth are store and load facts, respectively. Fig. 4c is the results of running the Datalog inference algorithm on those rules and facts. The example in Fig. 4a is called a *context-insensitive* pointer analysis because it does not distinguish between different objects, call sites and types in a class hierarchy. More precise context-sensitive pointer analyses take different kinds of context into consideration. For example, a *1-call-site-sensitive* analysis considers method call sites. A *1-object-*

sensitive analysis (similarly, *1-type-sensitive*) includes object allocation sites (types of objects allocated) as part of the context.

3 LIFTING DATALOG

In this section, we present our approach to lifting Datalog abstract syntax and the Datalog inference algorithm. We also formally state the correctness criteria for lifted Datalog inference, and present a correctness proof of our lifted algorithm.

3.1 Annotated Datalog Clauses

When analyzing a single software product, an initial set of facts is extracted from product artifacts, and analysis rules are applied to those facts, eventually adding newly inferred facts to the initial set. In the case of SPLs, a fact might be valid only in a subset of products, and not necessarily the entire product space. We have to associate a representation of that subset with each of the extracted facts. Similar to SPL annotation techniques, a *Presence Condition (PC)* is a succinct representation that can be used to annotate facts.

Facts annotated with PCs are called *lifted facts*, and are stored in a lifted Extensional Database – $\widehat{\text{EDB}}$. Given a feature expression ρ , we define $\widehat{\text{EDB}}|_\rho$ to be the set of facts from $\widehat{\text{EDB}}$ which only exist in the product set defined by ρ :

$$\widehat{\text{EDB}}|_\rho = \{f \mid (f, pc) \in \widehat{\text{EDB}} \wedge \text{sat}(pc \wedge \rho)\}$$


```

VarPointsTo(v1, h1) :- New(v1, h1).
VarPointsTo(v1, h2) :-
  Assign(v1, v2), VarPointsTo(v2, h2).
VarPointsTo(v1, h2) :-
  Load(v1, v2, f),
  VarPointsTo(v2, h1),
  HeapPointsTo(h1, f, h2).
HeapPointsTo(h1, f, h2) :-
  Store(v1, f, v2),
  VarPointsTo(v1, h1),
  VarPointsTo(v2, h2).

```

(a) Pointer Analysis Rules.

```

New("o1", "A").           // line 5
New("o2", "B").           // line 6
Assign("o3", "o2").       // line 7
Store("o2", "f", "o1").   // line 8
Load("r", "o3", "f").     // line 9

```

(b) Facts extracted from Listing 2.

```

VarPointsTo("o1", "A").
VarPointsTo("o2", "B").
VarPointsTo("o3", "B").
HeapPointsTo("B", "f", "A").
VarPointsTo("r", "A").

```

(c) Results of applying the rules to the extracted facts.

Figure 4: (a) Context-insensitive pointer analysis rules (simplistic), (b) input facts, and (c) output facts for program in Listing 2.

$$\begin{array}{c}
 \frac{C :- s_1, \dots, s_n \quad [\gamma]s_1 = f_1 \quad \dots \quad [\gamma]s_n = f_n \quad \forall (1 \leq i \leq n), f_i \in \text{EDB} \quad \gamma : V \rightarrow S}{[\gamma]C} \text{MP} \\
 \\
 \frac{C :- s_1, \dots, s_n \quad [\gamma]s_1 = f_1 \quad \dots \quad [\gamma]s_n = f_n \quad \forall (1 \leq i \leq n), (f_i, \textcolor{blue}{pc_i}) \in \widehat{\text{EDB}} \quad \gamma : V \rightarrow S}{([\gamma]C, \textcolor{blue}{pc_1} \wedge \dots \wedge \textcolor{blue}{pc_n})} \widehat{\text{MP}}
 \end{array}$$

Figure 5: Modus ponens for (a) Datalog clauses and (b) lifted Datalog clause inference (differences in blue).

When the Datalog inference algorithm is applied to annotated facts, we have to take the PCs attached to facts into account. Whenever the inference algorithm generates a new fact, we need to associate a PC to it. If f_{new} is generated from premises $f_1, f_2, \dots, f_n$, with PCs $pc_1, \dots, pc_n$, then pc_{new} attached to f_{new} should be the conjunction of the input PCs, i.e., $pc_1 \wedge \dots \wedge pc_n$. Intuitively, pc_{new} represents the set of products in which f_{new} exists, which is the intersection of the sets of products in which the premises exist.

To avoid having too many generated facts that are practically vacuous, we check pc_{new} for satisfiability. If it isn't satisfiable, then its corresponding fact exists in the empty set of products, i.e., non-existent. Those facts can be safely removed from $\widehat{\text{EDB}}$, potentially improving the performance of inference.

3.2 Lifted Inference Algorithm

Algorithm 2 takes a set of Datalog rules (IDB) and a set of annotated facts ($\widehat{\text{EDB}}$) as input, and returns all inferred clauses, annotated with their corresponding presence conditions. The structure of this algorithm is similar to that of Algorithm 1, with the exception of conjoining the presence conditions of the facts used in inference, and assigning the conjunction as the presence condition of the result. There are four cases for (c, pc_c) to consider: (1) if $\text{sat}(pc_c)$ is *False* (pc_c is not satisfiable), then this result is ignored because it doesn't exist in any valid product; (2) if $(c, pc_c) \in \widehat{\text{EDB}}$, then this result is also ignored because it already exists for the same set of products; (3) if $(c, pc_d) \in \widehat{\text{EDB}}$, where $pc_d \neq pc_c$, then (c, pc_d) is replaced with $(c, pc_d \vee pc_c)$ in $\widehat{\text{EDB}}$. This means we are expanding the already existing set of products in which c exists to also include the set denoted

by pc_c ; (4) if c doesn't exist at all in $\widehat{\text{EDB}}$, we add (c, pc_c) to it. For example, when the lifted inference algorithm is applied to the rules in Fig. 4a and annotated facts in Fig. 8, the result is the following:

```

VarPointsTo("o1", "A")           @ True.
VarPointsTo("o2", "B")           @ True.
VarPointsTo("o3", "A")           @ FA.
VarPointsTo("o3", "B")           @ !FA.
HeapPointsTo("B", "f", "A")      @ FB.
HeapPointsTo("B", "f", "B")      @ !FB.
VarPointsTo("r", "A")            @ FA.
VarPointsTo("r", "B")            @ !FA.

```

3.3 Correctness Criteria

When applying the lifted inference algorithm $\widehat{\text{infer}}$ to a set of rules IDB and a set of annotated facts $\widehat{\text{EDB}}$, we expect the result to be exactly the union of the results of applying infer to facts from each product individually. Moreover, each clause in the result of $\widehat{\text{infer}}$ has to be properly annotated (i.e., its presence condition has to represent exactly the set of products having this clause in their un-lifted analysis results).

Theorem 1. *Given an SPL $\mathcal{L} = (F, \Phi, D, \phi)$, a set of rules IDB , and a set of lifted facts $\widehat{\text{EDB}}$ annotated with feature expressions over F :*

$$\forall (\rho \in \text{Conf}(\mathcal{L})), \widehat{\text{infer}}(\widehat{\text{EDB}})|_\rho = \text{infer}(\widehat{\text{EDB}}|_\rho)$$

Proof.

Forward implication: $C \in \widehat{\text{infer}}(\widehat{\text{EDB}})|_\rho \implies C \in \text{infer}(\widehat{\text{EDB}}|_\rho)$

Input: $IDB, \widehat{EDB}$
Result: $\widehat{EDB} \cup \{\text{annotated inferred clauses}\}$

```

1 repeat
2   fixpoint  $\leftarrow$  True;
3   foreach  $(C := s_1, \dots, s_n) \in IDB$  do
4     foreach  $(\gamma, (f_1, pc_1), \dots, (f_n, pc_n)), (f_i, pc_i) \in$ 
        $\widehat{EDB}, [\gamma]s_i = f_i$  do
5        $pc_c \leftarrow pc_1 \wedge \dots \wedge pc_n$ ;
6       if  $\text{sat}(pc_c)$  then
7         if  $([\gamma]C, pc_c) \notin \widehat{EDB}$  then
8           fixpoint  $\leftarrow$  False;
9           if  $\exists pc_d, ([\gamma]C, pc_d) \in \widehat{EDB}$  then
10             $pc_c \leftarrow pc_c \vee pc_d$ ;
11             $\widehat{EDB} \leftarrow \widehat{EDB} - \{([\gamma]C, pc_d)\}$ ;
12          end
13           $\widehat{EDB} \leftarrow \widehat{EDB} \cup \{([\gamma]C, pc_c)\}$ ;
14        end
15      end
16    end
17  end
18 until fixpoint;
19 return  $\widehat{EDB}$ ;

```

Algorithm 2: Lifted inference algorithm $\widehat{\text{infer}}$ (differences against Algorithm 1 in blue).

By structural induction over the derivation tree of C :

Base Case: $(C, pc) \in \widehat{EDB}$, where $\text{sat}(pc \wedge \rho)$. Then $C \in \widehat{EDB}|_\rho$ (by definition of restriction operator). Since inputs are already included in the output of $\widehat{\text{infer}}$, $C \in \widehat{\text{infer}}(\widehat{EDB}|_\rho)$.

Induction Hypothesis: Given a rule $R = C := s_1, \dots, s_n$, and a variable assignment γ ,

$$\forall (1 \leq i \leq n) : [\gamma]s_i \in \widehat{\text{infer}}(\widehat{EDB})|_\rho \implies [\gamma]s_i \in \widehat{\text{infer}}(\widehat{EDB}|_\rho)$$

Induction Step: C is derived by $\widehat{MP}$ (Fig. 5) from rule R . Since all the premises of C are in $\widehat{\text{infer}}(\widehat{EDB}|_\rho)$ (induction hypothesis), then so is C ($\widehat{MP}$).

Backward implication: $C \in \widehat{\text{infer}}(\widehat{EDB}|_\rho) \implies C \in \widehat{\text{infer}}(\widehat{EDB})|_\rho$

By structural induction over the derivation tree of C :

Base Case: Assume $(C, pc) \in \widehat{EDB}$, for some pc , where $\text{sat}(pc \wedge \rho)$. Then $(C, pc) \in \widehat{\text{infer}}(\widehat{EDB})$ (input included in output of $\widehat{\text{infer}}$). Since $pc \wedge \rho$ is satisfiable, then $C \in \widehat{\text{infer}}(\widehat{EDB})|_\rho$ (definition of restriction).

Induction Hypothesis: Given a rule $R = C := s_1, \dots, s_n$, and a variable assignment γ ,

$$\forall (1 \leq i \leq n) : [\gamma]s_i \in \widehat{\text{infer}}(\widehat{EDB}|_\rho) \implies [\gamma]s_i \in \widehat{\text{infer}}(\widehat{EDB})|_\rho$$

Induction Step: C is derived by MP (Fig. 5) from rule R . Since all the premises of C are in $\widehat{\text{infer}}(\widehat{EDB})|_\rho$ (induction hypothesis), then so is C ($\widehat{MP}$). $\square$

The lifted Datalog inference algorithm, together with input facts annotated with presence conditions, are the foundations for variability-aware Datalog analyses.

4 IMPLEMENTATION

In this section, we describe the design and implementation modifications we made to the Soufflé Datalog engine to make it variability-aware (Sec. 4.1). We also explain the method for lifting the Doop pointer and taint analysis framework, which runs on top of Soufflé. We outline the modifications we made to the Doop fact extractor (extracting facts from Java bytecode) to make it variability-aware (Sec. 4.2.1). Finally, we outline the design of a C-language fact extractor, and a set of Datalog analyses for C programs (Sec. 4.2.2).

The overall architecture of the two analysis pipelines is outlined in Fig. 6. The Doop pipeline takes "Java Bytecode" files as input, and the "Java Fact Extractor" (a part of the Doop framework) extracts Datalog facts from them. Those facts, together with the "Java Rules", are sent to the "V-Soufflé" variability-aware Datalog engine, which generates the results.

The C-language pipeline takes "C Program" source files as input. Because variability in those files is annotated using the C Pre-Processor (CPP), we use "TypeChef" to parse them. TypeChef generates "V-AST" (Variability-aware Abstract Syntax Tree), and "V-CFG" (Variability-aware Control Flow Graph) files, from which the "C Fact Extractor" extracts Datalog facts. The code analyses are implemented as a database of Datalog "C Rules" which are passed to V-Soufflé together with the extracted facts. V-Soufflé generates the analysis results.

4.1 Lifting Soufflé

Soufflé [9] is a highly optimized open source Datalog engine. Our modified variability-aware version of Soufflé¹ (V-Soufflé) implements our lifted Datalog inference algorithm $\widehat{\text{infer}}$ (Sec. 3).

As seen in Fig. 7, a Soufflé program (a Datalog IDB) is passed to a "Datalog to RAM Compiler", which parses the program and translates it into a "Relational Algebra Machine (RAM) Program". *RAM* is a Relational Algebra language [26], in addition to a fixed-point looping operator. Depending on a command-line argument, Soufflé then either interprets the RAM program on the fly using the "RAM Interpreter", or synthesizes a "C++ Program" that is semantically equivalent to the RAM program using the "C++ Synthesizer". The C++ program can be then compiled using an off-the-shelf "C++ Compiler" into a native "Executable". Native executables are usually at least an order of magnitude faster than interpreted RAM programs [9]. In this paper, we only lift the Soufflé interpreter, which involved modifying the Datalog to RAM Compiler, and the RAM Interpreter components.

Datalog to RAM Compiler. At the syntax level, we extended the Soufflé language with *fact annotations* – propositional formulas prefixed with '@'. The Soufflé parser is extended with a syntactic category for propositional formulas. AST nodes for facts are extended with a PC field, with a default value of `True`. Propositional variables are added to

1. Available online at <https://github.com/ramyshahin/souffle>

Figure 6: End-to-end architecture for both the Doop-based Java analysis pipeline and the C-language analysis pipeline. Data files and databases are inputs/outputs to logic components (programs and Datalog rules). Components created by us have a red border; components modified by us have a light-blue border.

Figure 7: Soufflé architecture. Components modified for V-Soufflé are have a light-blue border.

a separate symbol table from the one with Soufflé identifiers.

As a part of compiling Soufflé programs into RAM, we turn syntactic presence conditions into Binary Decision Diagrams (BDDs). We use the CUDD library [27] as a BDD engine, and on top of it maintain a map from textual presence conditions to their corresponding canonical BDDs. As defined in *infer*, when facts are resolved with a rule, the conjunction of their PCs becomes the conclusion’s PC.

RAM Interpreter. The RAM Interpreter takes a RAM program as input, and executes it on the fly. This involves both running the Datalog forward-chaining inference algorithm (*infer*) to infer new facts, and applying relational algebra operators such as selection, projection and joins. Soufflé implements several indexing and query optimization techniques to improve inference time. To keep our modifications independent of those optimizations, we add the presence condition as a field opaque to the query engine. We only

manipulate this field as a PC when performing clause resolution, which takes place at a higher level than the details of indexing and query processing. This way we avoid touching relatively complex optimization code, while preserving the semantics of our lifted inference algorithm.

Some relational features of Soufflé were not lifted. For example, aggregation functions (sum, average, max, min, etc...) still return singleton values. None of those functions is used by Doop or our C-language analyses on lifted facts, so this does not affect the correctness of our results. We plan to address this general limitation in the future.

4.2 Fact Extraction

Datalog analyses are not applied directly to programs. Instead, Datalog facts are extracted from the source code, and those facts are the input upon which a Datalog analysis operates. Extracted facts are syntactic in the sense that they correspond to syntactic structures. Variability-aware facts have presence conditions associated to them, denoting the set of products in which a given fact exists. In particular, since several syntactic tokens can contribute to a fact, intuitively, the fact exists only if all the tokens contributing to it exist. In other words, the presence condition of a fact is the conjunction of the presence conditions of all of its tokens.

In this subsection, we outline our approaches to variability-aware fact extraction from Java bytecode and from C-language source code.

4.2.1 Lifting Doop

As a part of our evaluation of the Datalog lifting approach outlined in Sec. 3, we modified the Doop [11] Datalog-based pointer analysis framework², which uses Soufflé as its underlying Datalog engine. Fig. 6 shows the Doop architecture together the rest of the analysis pipeline. Doop is an

2. Available online at <https://bitbucket.org/rshahin/doop>

```

New("o1", "A")           @ True. // line 5
New("o2", "B")           @ True. // line 6
Assign("o3", "o1")        @ FA.   // line 8
Assign("o3", "o2")        @ !FA.  // line 10
Store("o2", "f", "o1")    @ FB.   // line 13
Store("o2", "f", "o2")    @ !FB.  // line 15
Load("r", "o3", "f")      @ True. // line 17

```

Figure 8: Annotated facts extracted from Listing 1.

```

int* main(){
    int* x,y;
    x = malloc(sizeof(int));
    #ifdef CREATE
    y = malloc(sizeof(int));
    #ifdef ASSIGN
    x = y;
    #endif // end ASSIGN
    #endif // end CREATE
    return x;
}

```

Figure 9: A simple C program with variability.

extensible family of pointer and taint analyses implemented as Datalog rules. In addition, it includes a Java bytecode fact extractor. Doop users select a particular analysis among the available analyses through a command-line argument. The rules corresponding to the chosen analysis (the IDB), together with the extracted facts (the EDB), are then passed to Soufflé.

Since Doop extracts syntactic facts, we need to identify the PCs of each of the syntactic tokens contributing to a fact, and associate the conjunction of those PCs as the fact PC. We had to do this for each type of fact extracted by Doop. The fact PC is just added to a fact as a trailing PC field, prefixed with '@'. Facts with no PC field are assumed to belong to all products (an implicit PC of `True`).

Our Doop modifications were limited to the fact extractor. None of the Doop Datalog rules were changed. Our fact extraction modifications were scattered because Doop implements extractors for different kinds of facts separately. However, all of our changes were systematic and non-invasive.

All the Java benchmarks we used to evaluate lifted Doop were annotated using CIDE (Sec. 2.1). The presence condition of a fact is the conjunction of the feature expressions of each of the tokens contributing to it. We use line numbers to find the feature expression of each token. Java bytecode tokens already have their respective source code line numbers stored as metadata. We had to extract the coloring information for each line of code from the CIDE system, and map each line to the feature expression corresponding to its highlighting color. A line's feature expressions is then assigned to the tokens belonging to that line.

4.2.2 C-language Fact Extraction

In this section, we discuss our C-language fact extraction framework and its implementation – see Fig. 6 for the high-level architecture. Unlike our Java fact extractor, we

Figure 10: V-AST of Fig. 9 generated similar to TypeChef output. Some nodes were not expanded for readability.

designed our C fact extractor from scratch because modern fact extractors for C predominantly rely on LLVM IR [?] but compiling C to LLVM IR loses variability. Performing analysis directly on C instead of LLVM IR is significantly more difficult due to the more complex syntax and semantics of C.

Variability Annotations in C. Variability can be added to C programs using the `#ifdef` directive. Fig. 9 shows a simple program with two features, `ASSIGN` and `CREATE`. Variability can be nested; in this example, both features need to be set to `True` in order to compile code under the `ASSIGN` section.

V-AST and V-CFG Generation. We use TypeChef [5] to generate *Variability Aware Abstract Syntax Tree* (V-AST) and *Variability Aware Control Flow Graph* (V-CFG). A V-AST is a standard AST representation of the program, with additional nodes to represent presence conditions. A V-CFG is a standard representation of the control flow of a program, with feature presence condition annotations added to each node.

Fig. 11 shows the V-CFG generated by TypeChef for the program in Fig. 9. The format of the file for nodes consists the lines starting with the character “N”, followed by the identifier of the node, line number or –1 if not relevant, the function where the node was located, and the variability condition. For edges, it consists of the line starting with the character “E”, followed by the IDs of the predecessor and successor nodes, and the variability condition of the edge.

V-AST Processing. Algorithm 3 shows the procedure for generating variability-aware facts from a V-AST. It uses *FactDef* which is a pair of functions *fits()* and *gets()*, to define a requirement for a fact to extract, such as a variable assignment, and generate that fact from an AST node if it fits the requirement. Fig. 10 shows an example V-AST generated by TypeChef of the program in Fig. 9. The *PresenceConditionNodes* in the V-AST correspond to the `#ifdef` directives from the program. Fig. 12 shows the facts generated from the V-AST in Fig. 10. To generate these facts, the *FactDef* functions were assigned to extract facts for Pointer Analysis,


```

N;1669;statement;-1;x = y::main;(definedEx(CREATE) && definedEx(ASSIGN))
N;1940;statement;10;return x::main;1
E;1669;1940;(definedEx(CREATE) && definedEx(ASSIGN))
N;3588;function;-1;main;1
E;1940;3588;1
N;1656;statement;5;(y = malloc(sizeof(int))::main;definedEx(CREATE)
E;1656;1669;(definedEx(CREATE) && definedEx(ASSIGN))
E;1656;1940;(definedEx(CREATE) && !definedEx(ASSIGN))
N;6460;statement;3;(x = malloc(sizeof(int))::main;1
E;6460;1656;definedEx(CREATE)
E;6460;1940;!definedEx(CREATE)
N;2138;statement;-1;int* x,y::main;1
E;2138;6460;1
E;3588;2138;1

```

Figure 11: V-CFG Generated by TypeChef of the program in Fig. 9. Lines that start with “N” and “E” identify a node and an edge respectively. The numbers represent identifiers for nodes, and identifiers of both nodes for edges.

```

allocHeap("x", "1").
allocHeap("y", "2") @ CREATE.
pointsTo("x", "y") @ (CREATE/\ASSIGN).

```

Figure 12: Extracted pointer analysis facts from Fig. 10.

```

dominates("2138", "6640").
dominates("2138", "1659") @ (CREATE).
dominates("2138", "1669") @ (CREATE/\ASSIGN).
dominates("2138", "1940").
dominates("1659", "1669") @ (CREATE/\ASSIGN).
dominates("1659", "1940") @ (CREATE).
dominates("1669", "1940") @ (CREATE/\ASSIGN).

```

Figure 13: Extracted variable node dominator relationship facts from Fig. 11. The first parameter is the identifier of the dominating node and the second is the dominated node. The presence conditions were simplified where possible for readability; the simplification is not done as part of the algorithm.

described in Sec. 5.2. MC: not clear how FactDef is pointer analysis. [paragraph updated](#)

We assume that the V-AST generated by TypeChef is correct and that FactDef correctly identifies nodes of interest. Our correctness criterion that the presence condition of the extracted fact is `True` if and only if the presence condition of all tokens that contributed to that fact is `True`. In this case, since the fact is generated from a single AST-subtree using FactDef on line 8, we can guarantee that the presence condition of that fact is `True` iff the presence condition of the subtree is `True`; this is because `currCond` on line 3 is set to the conjunction of all parent present condition nodes. Finally, the algorithm assumes that there is no variability within a statement. To handle it, we first use TypeChef to generate a V-CFG with TypeChef – this process automatically splits statements that contain variability – and then parse ASTs of those statements.

V-CFG Processing. Algorithm 4 shows the procedure we use to extract facts from a V-CFG. This algorithm relies on an external function `cfgAnalysis()`. For a given CFG, this

Input: ASTNode, Presence condition, Fact Definition

Mutates: Facts

```

1 if ASTNode instanceof(PresenceConditionNode) then
2   thisCond ← ASTNode.condition()
3   currCond ← conjunct(currCond, thisCond)
4   foreach child of ASTNode do
5     | recurse(child, currCond, factDef)
6   end
7 else if factDef.fits(ASTNode) then
8   /* Subtree is statement that corresponds to a fact */
9   fact ← FactDef.get(ASTNode)
10  facts.append((fact, currCond))
11  foreach child of ASTNode do
12    | recurse(child, currCond, factDef)
13  end
14 else
15   foreach child of ASTNode do
16     | recurse(child, currCond, factDef)
17   end

```

Algorithm 3: Generating facts from V-AST. FactDef is an input structure that defines methods `fits()` and `get()`. The former checks if a fact we are looking for can be extracted from the AST Node, and the latter generates the fact from an applicable node.

function returns a list of desired relationships, and a list of nodes related to each relationship. We used OCamlgraph³ to assist with control-flow analysis, specifically, with identifying node domination relationships. The `cfgAnalysis` procedure can be a wrapper around OCamlGraph or a different directed graph analysis utility. We discuss the analyses we used in detail in Sec. 5.2. Fig. 13 shows simple dominator relationship facts extracted from the program in Fig. 9, using Alg. 4. The first parameter of every fact represents the dominating node, while the second the dominated node. To perform this analysis, the `cfgAnalysis` helper would be set to a function that finds all dominator pairs in a directed graph. [paragraph updated](#)

3. <http://ocamlgraph.lri.fr/>

Input: V-CFG, CFG Analysis Function

Mutates: Facts

```

1 factNodesPairs ← cfgAnalysis(V-CFG)
2 foreach fact,nodes in factNodePairs do
3   conditions ← emptyList()
4   foreach node in nodes do
5     conditions.append(node.condition)
6   end
7   presenceCond ← conjunct(conditions)
8   facts.append((fact, presenceCond))
9 end

```

Algorithm 4: Generating facts from V-CFG. The algorithm uses a helper function *cfgAnalysis()* that performs the desired directed graph analysis returning pairs of extracted facts and the nodes relating to them.

We assume that the V-CFG generated by TypeChef is correct and that *cfgAnalysis()* works as described. Our correctness criterion is that the presence condition of the extracted fact is *True* if and only if the presence conditions of all tokens that contributed to that fact are *True*.

The presence conditions of all CFG nodes are collected on line 5, and then conjoined to generate a fact on line 7. Therefore, the presence condition of the generated fact evaluates to *True* only if the presence conditions of all nodes are *True*.

5 EVALUATION

We evaluate the performance of our lifted version of Soufflé using two sets of benchmarks: Three Doop analyses applied to five Java benchmark product lines (previously used in the evaluation of other lifted analyses [2], [28]), and the three hand-crafted C-language analyses outlined in Sec. 5.2 applied to the BusyBox⁴ product line.

The primary goal of our experiments is to compare the performance of lifted analyses applied to the SPLs to that of running the corresponding product-level analyses on each of the valid configurations individually. Since the number of valid product configurations for some benchmarks is relatively big, it is neither practical nor particularly useful to enumerate all of the valid products and analyze them. Instead, for each SPL, we run the product-level analysis on two code-base subsets: the base code common across all variants, and the 150% representation (the whole SPL code-base, implementing all feature behaviors). Although those two extremes are not necessarily valid products, they are the lower bound and the upper bound in terms of code size, and averaging over them gives an "average" valid product approximation. The *expected brute-force performance* is the average valid product performance (*P-Avg*) multiplied by the number of valid configurations.

We split our evaluation into two parts: fact extraction and inference, and evaluate performance in terms of both the processing time and space (size of the fact database in KiloBytes(KB)). Our primary research questions are:

4. <https://www.busybox.net/>

Table 1: Java product lines used for evaluation.

Benchmark	Size (KLOC)	Features	Valid Configurations
BerkeleyDB	70	42	8,759,844,864
GPL	1.4	21	4,176
Lampiro	45	18	2,048
MM08	5.7	27	784
Prevayler	7.9	5	32

RQ1: How do fact extraction time (and size of the extracted fact database) of lifted analyses compare to brute-force fact extraction?

RQ2: How do the Soufflé inference time, and the size of the inferred database, of lifted analyses compare to brute-force analysis?

RQ3: How consistent are the savings in inference time and size of the fact database across different languages, and across different annotation mechanisms?

5.1 Java Benchmarks

In this subsection we present a set of evaluation experiments on three analyses from the Doop Java pointer analysis framework. The lifted analyses are applied to five different Java benchmark product lines.

5.1.1 Models and Methods

For each of the Java benchmarks, Table 1 lists its size (in thousands of lines of code), number of features, and number of valid configurations according to its feature model. For example, BerkeleyDB is about 70,000 lines of code, comprised of 42 features, and has about 8.76 billion valid product configurations. We evaluate three Doop analyses: context-insensitive pointer analysis (*insens*), one-type heap-sensitive pointer analysis (*1Type+Heap*), and one-call-site heap-sensitive taint analysis (*Taint-1Call+Heap*). For taint analysis, we use the default sources, sinks, transform and sanitization functions curated in Doop for the JDK and Android [23]. All experiments were performed on a Quad-core Intel Core i7-6700 processor running at 3.4GHZ, with 16GB RAM and hyper-threading enabled, running 64-bit Ubuntu Linux (kernel version 4.15).

Pointer and taint analyses work on the whole program, including library dependencies. Since general-purpose libraries usually do not have any variability, the comparison between lifted and single-product analyses is independent of them. Moreover, time spent in analyzing library code, and space taken by their facts, might skew the overall results. We restrict our experiments to application code and direct dependencies only using the Doop command-line argument `"-Xfacts-subset APP_N_DEPS"`.

Doop extracts its facts from Java byte-code. However, SPL annotation techniques work at the source-code level. Feature selection usually takes place at compile-time, which means an SPL codebase is compiled into a single product. To get around this limitation, we had to choose benchmarks that only have *disciplined annotations* [16], in the sense that adding or removing an annotation preserves the syntactic correctness of the 150% representation. This is not a limitation of our lifted inference algorithm though.

Figure 14: Doop context-insensitive fact extraction speedup and DB savings factors: SPL vs. average product.

The benchmarks we chose are annotated using CIDE [16], which uses different highlighting colors as presence conditions. We had to extract this color information from CIDE, together with the mapping from colors to locations of tokens (line and column number) in source files. Our fact extractor uses byte-code symbol information to locate tokens, and assign their presence conditions based on CIDE colors.

5.1.2 Fact Extraction

Table 2 summarizes the "average" performance of product-level fact extraction (*P-Avg*) and that of the lifted fact extraction for the entire product line (*SPL*). For each of the three analyses, we compare fact extraction time (in milliseconds) and the size of the extracted database (in KB). For example, for context-insensitive analysis, average fact extraction time of a single product of Prevayler is 1,416ms, and the average size of the extracted fact database is 3,230KB. On the other hand, extracting facts from the whole Prevayler SPL at once takes 1,554ms, and the extracted fact database is 4,407KB. The difference between *P-Avg* Time and *SPL* Time is very small for all three analyses and five benchmarks, which is expected since extraction is syntactic and thus its time is proportional to code-base size, not the number of features. Size of the extracted database is noticeably bigger for lifted extraction (DB *SPL* columns) because lifted facts are augmented with presence conditions.

To evaluate the savings attributed to lifted fact extraction compared to brute-force extraction in terms of time and space, we compute the speedup and space saving factors ($P\text{-Avg} * |\text{Conf}(\mathcal{L})| / \text{SPL}$). Fig. 14 shows a log-scale bar graph of lifted fact extraction speedup and space savings for context-insensitive analysis. The other two analyses exhibit a similar trend and are omitted here. The figure shows that the time and space savings are proportional to the number of valid configurations of the product line. For example, Lampiro has 2048 valid configurations, and its lifted fact extraction is 2020 times faster than brute-force, with a database 2045 times smaller than the total space of brute-force databases. On the other hand, Prevayler has only 32 valid configurations, with an *insens* lifting speedup factor of 29, and a space savings factor of 23. Since different analyses typically require different facts, the size of the fact

Figure 15: Doop context-insensitive Inference speedup and DB savings factors: SPL vs average product.

database also varies from one analysis to another. Experimental results do not show a direct correlation between an analysis and the size of its fact database. For example, in Lampiro, the *Taint-1Call+Heap* databases are significantly bigger than those of *1Type+Heap*. BerkeleyDB, on the other hand, exhibits the opposite trend.

5.1.3 Inference

Table 3 summarizes the performance of lifted analyses on the entire product line (*SPL*) and that of product-level analyses on an average product (*P-Avg*). For example, running *1Type+Heap* on an average MM08 product, inference is estimated to take 4,596ms, resulting in a database of 8,788 KB. Running the same analysis on the entire MM08 product line though takes 8,788 ms, resulting in a 13,021 KB database. Fig. 15 is a log-scale bar graph of the speedup factor and the DB space savings factor for *insens*. Speedup and space savings trends are again proportional to the number of valid configurations. For example, for BerkeleyDB, lifted *insens* is about 7.4 billion times faster than brute-force, with a DB 5.6 billion times smaller. All three analyses show similar speedup and disk space savings trends.

Recall that the theoretic bottleneck of the lifted inference algorithm (Algorithm 2) is the satisfiability checks performed when conjoining two PCs. Since propositional satisfiability is *NP-complete*, we wanted to evaluate whether it is a bottleneck in practice. While SAT checks are not required to maintain correctness of the lifted inference algorithm, we perform them in order to avoid generating spurious facts that do not exist in any product. An UNSAT presence condition denotes an empty set of products, but what about PCs denoting sets of invalid product configurations? The *Feature Model (FM)* of a product line specifies which product configurations are valid and which are not. If a fact belongs only to a set of configurations excluded by the FM, then this fact can be removed. Removing spurious facts saves DB space, and, more importantly, it keeps the set of facts searched by the inference algorithm as small as possible, improving the overall performance. We study the impact of SAT checking and using the FM below.

RQ2.1: How much does SAT checking contribute to the processing time of the lifted Datalog engine?

Table 2: Doop fact extraction time (in ms) and DB size (in KB): Average Product (P-Avg) vs SPL for all three analyses.

	<i>insens</i>				<i>1Type+Heap</i>				<i>Taint-1Call+Heap</i>			
	<i>Time(ms)</i>		<i>DB(KB)</i>		<i>Time(ms)</i>		<i>DB(KB)</i>		<i>Time(ms)</i>		<i>DB(KB)</i>	
	P-Avg	SPL	P-Avg	SPL	P-Avg	SPL	P-Avg	SPL	P-Avg	SPL	P-Avg	SPL
BerkeleyDB	5,136	4,541	31,892	49,725	4,651	4,809	71,536	122,922	4,647	4,667	64,497	112,060
GPL	816	814	175	409	782	876	245	593	789	802	188	462
Lampiro	4,413	4,475	41,100	41,170	4,425	4,145	149,521	149,686	4,237	4,436	230,035	230,370
MM08	1,226	1,364	1,921	3,259	1,250	1,372	3,878	6,990	1,255	1,252	4,234	7,829
Prevayler	1,416	1,554	3,230	4,407	1,453	1,454	5,882	8,630	1,502	1,404	3,917	5,534

Table 3: Doop inference time (in ms) and inferred DB size (in KB): Average Product (P-Avg) vs SPL for the three analyses.

	<i>insens</i>				<i>1Type+Heap</i>				<i>Taint-1Call+Heap</i>			
	<i>Time(ms)</i>		<i>DB(KB)</i>		<i>Time(ms)</i>		<i>DB(KB)</i>		<i>Time(ms)</i>		<i>DB(KB)</i>	
	P-Avg	SPL	P-Avg	SPL	P-Avg	SPL	P-Avg	SPL	P-Avg	SPL	P-Avg	SPL
BerkeleyDB	9,184	10,810	141,728	200,655	13,598	17,273	318,349	483,206	17,479	21,474	285,473	443,737
GPL	4,422	4,517	2,033	3,528	4,794	4,718	3,237	5,675	8,999	8,861	2,500	4,450
Lampiro	8,264	8,111	245,933	246,285	21,372	20,725	980,689	981,549	44,365	45,996	1,393,038	1,394,826
MM08	4,596	4,720	8,788	13,021	5,106	5,142	19,058	29,453	9,340	9,306	18,302	29,383
Prevayler	4,908	5,334	15,856	19,808	5,852	6,013	29,605	38,747	9,785	9,717	20,611	26,279

Table 4: Doop SAT vs. noSAT. Time in milliseconds, Inferred DB in KB.

	<i>insens</i>				<i>1Type+Heap</i>				<i>Taint-1Call+Heap</i>			
	<i>Time(ms)</i>		<i>DB(KB)</i>		<i>Time(ms)</i>		<i>DB(KB)</i>		<i>Time(ms)</i>		<i>DB(KB)</i>	
	SPL	noSAT	SPL	noSAT	SPL	noSAT	SPL	noSAT	SPL	noSAT	SPL	noSAT
BerkeleyDB	10,810	10,879	49,725	53,151	17,273	17,784	122,922	143,720	21,474	21,920	112,060	128,427
GPL	4,517	4,496	409	472	4,718	4,667	593	765	8,861	8,812	462	595
Lampiro	8,111	8,105	41,170	41,170	20,725	21,224	149,686	149,710	45,996	47,132	230,370	230,370
MM08	4,720	4,689	3,259	3,762	5,142	5,118	6,990	8,775	9,306	9,270	7,829	9,082
Prevayler	5,334	5,050	4,407	4,638	6,013	5,940	8,630	9,392	9,717	9,903	5,534	5,869

Figure 16: Doop inference time: SPL vs. SPL with SAT checking disabled for all three analyses.

Table 4 summarizes the performance of our lifted analyses and the same analyses with SAT checking disabled (noSAT). Fig. 16 and Fig. 17 show the noSAT-associated speedup and database size savings, respectively. Recall that we represent PCs using BDDs. SAT checking over BDDs is a constant-time operation [18]. Since conjoining and disjoining BDDs can take exponential time, we disable all BDD operations, keeping only the textual representation of PCs. A speedup factor below 1.0 means that disabling SAT checks slows down inference. This is what we observed for most of the benchmarks. We believe that the slowdown is because of the use of textual representation of PCs which resulted in a much bigger PC table, with slower lookup times. We also do not see any DB savings because non-canonically represented

Figure 17: Doop inferred database size (KB): SPL vs. SPL with SAT checking disabled for all three analyses.

PCs tend to be longer than BDD-based ones, resulting, on average, in more characters (and bytes) per PC. We note that the number of features is relatively low in all of our benchmarks. BDD-based SAT solving is known to perform well when the number of propositional variables is this small. With product lines of hundreds or thousands of features, it is possible that noSAT might result in performance improvements.

RQ2.2: What is the effect of taking the feature model (FM) of an SPL into consideration when running Datalog variability-aware analyses, in terms of inference time and

Table 5: Doop SPL vs. SPL+FM. Time in milliseconds, Inferred DB in KB.

	<i>insens</i>				<i>1Type+Heap</i>				<i>Taint-1Call+Heap</i>			
	<i>Time(ms)</i>		<i>DB(KB)</i>		<i>Time(ms)</i>		<i>DB(KB)</i>		<i>Time(ms)</i>		<i>DB(KB)</i>	
	SPL	SPL+FM	SPL	SPL+FM	SPL	SPL+FM	SPL	SPL+FM	SPL	SPL+FM	SPL	SPL+FM
BerkeleyDB	10,810	11,693	49,725	721,927	17,273	22,276	122,922	1,859,320	21,474	24,140	112,060	1,625,278
GPL	4,517	4,587	409	9,447	4,718	4,728	593	13,809	8,861	8,918	462	10,644
Lampiro	8,111	8,528	41,170	325,848	20,725	22,283	149,686	1,283,639	45,996	48,688	230,370	1,843,151
MM08	4,720	4,761	3,259	69,017	5,142	5,288	6,990	158,732	9,306	9,476	7,829	158,330
Prevayler	5,334	5,169	4,407	8,825	6,013	5,984	8,630	17,564	9,717	9,977	5,534	11,394

Table 6: The number of inferred facts with and without the Feature Model (FM) on Doop.

	<i>insens</i>		<i>1Type+Heap</i>		<i>Taint-1Call+Heap</i>	
	SPL	SPL+FM	SPL	SPL+FM	SPL	SPL+FM
BerkeleyDB	200,655	200,650	483,206	483,201	443,737	443,732
GPL	3,528	3,128	5,675	4,821	4,450	3,800
Lampiro	246,285	246,221	981,549	980,823	1,394,826	1,394,825
MM08	13,021	13,021	29,453	29,452	29,383	29,381
Prevayler	19,808	19,808	38,747	38,747	26,279	26,279

Figure 18: Doop inference time: SPL vs. SPL with FM for all three analyses.

Figure 19: Doop inferred database size (KB): SPL vs. SPL with FM for all three analyses.

DB size?

Table 5 compares the performance of our lifted analyses against the same analyses using the feature model (SAT+FM). SAT+FM entails conjoining the feature model to each PC before performing the satisfiability check. If the PC encodes a set of products excluded by the FM, the conjunction is unsatisfiable. Fig. 18 and Fig. 19 show the SAT+FM-associated speedup and space savings, respectively. For most of the experiments, using the FM results in slowdowns and larger DBs. FM usage reduces the number

of inferred facts, as observed in Table 6, but the reduction is relatively small. On the other hand, PCs now conjoined with the FM are more complex, taking longer to construct (hence the performance penalty), and more bytes to store (hence the bigger DBs).

5.2 C-Language Benchmarks

In this subsection, we present the Datalog rules we used as our benchmarks. We then outline the evaluation process and present the evaluation results.

5.2.1 Models and Methods

We have implemented three Datalog analyses which we describe below.

Context-insensitive pointer analysis. Similar to context-insensitive pointer analysis in Doop, the goal is to statically identify the set of heap objects each pointer might be pointing to. Extracted facts include variable assignments, and functions returning pointers. As this is a context insensitive analysis, all variables that are assigned a return value from the same function are assumed to be pointing to the same location in memory, except for memory allocation functions such as `malloc`.

Control Flow Graph Reducibility. A Control Flow Graph (CFG) is said to be *reducible* if the graph has no cycles after all back-edges are removed. An edge from node *a* to node *b* is said to be a *back-edge* if *b* dominates *a* [29]. This analysis identifies all possible cycles in the graph using Johnson’s algorithm [30]. We used Pierto Abate’s functional OCaml implementation [31] for fact extraction. This analysis produces a large number of facts as they include all dominance relationships between all nodes in the CFG. For example, in a purely linear program, each additional statement would be dominated by every previous statement, resulting in an exponential growth in number of facts with every new statement.

Definition-usage chains. *Definition-usage chains analysis* (def-use) finds pairs of statements in the CFG where a variable is assigned (or defined) and then referenced [32]. We used Datalog implementation of this analysis from the

(a) Def-use storage usage.

(b) Pointer analysis storage usage.

(c) Reducibility storage usage.

Figure 20: C benchmarks storage usage.

(a) Def-use inference time.

(b) Pointer inference time.

(c) Reducibility inference time.

Figure 21: C benchmarks inference and extraction time.

Figure 22: Distribution of features in BusyBox files.

Soufflé tutorials⁵. Extracted facts for this analysis include all edges in the CFG, variable loads (references), and variable stores (assignments). **MC: smth concrete**

We evaluate each of the three analyses on 229 C files from the Busybox product line of Unix command-line tools (version 1.18.5). Out of 522 Busybox files, 33 were excluded due to errors when parsing C files with the TypeChef, 12 due to parsing errors of Typechef’s V-AST by our fact extractor, and over 260 due to lack of variability in the V-CFG. We also excluded the `CONFIG_NOMMU` feature as we found that it appears on every node of the V-CFG generated by TypeChef.

We compare our lifted approach against the estimate of brute-force analysis. We estimate brute force performance by applying the analyses to the largest and smallest fact-base of each source file, taking the average of the two, and multiplying the average by the number of possible products. Largest and smallest fact-bases are defined as the AST and CFG with all features included and all features excluded (leaving only the base system) respectively.

Fig. 22 represents the distribution of features-per-file for the files we analyzed. The majority of files have between zero and two features and thus we expect the performance of our analysis to be similar to our estimate of brute force. However, 90 files have three or more features, and we expect to see our approach yield a more substantial improvement in these cases.

We performed the experiments on a server with a 3GHz 32-core 64-bit processor, 128 GB of RAM, running Ubuntu Linux 18.04.

5.2.2 Fact Extraction

Fig. 20.a shows input storage usage (in bytes) for Pointer Analysis of our approach compared to the estimate of brute force. Input size is the size of the fact database generated by fact extraction. Our approach demonstrates clear constant scaling with pointer analysis (Fig. 20b). Brute force approach demonstrated exponential scaling with every analysis with respect to the number of features in a file. We can also see that storage savings of our approach are most noticeable for files with over five features, while space overhead is minimal for files with only two features. The comparisons

for DefUse and reducibility analyses are analogous. **MA: I meant that the delta is the same, but this kinda sounds awkward**

Fig. 21.a shows Pointer Analysis performance (in ms) of our approach compared to the estimate of brute force. Our approach demonstrated clear, near-constant growth for in terms of fact extraction time, while brute force demonstrated exponential growth. The comparisons for DefUse and reducibility analyses are analogous. The results are most consistent **MA: I meant it has very little noise, not sure what’s the best way to put it** for pointer analysis because it does not rely on the dominator relationship, the number of dominator relationships between CFG nodes can vary vastly depending on the structure and size of the file, resulting in much larger running times for some files independent of the number of features.

5.2.3 Inference

Fig. 21.a shows Pointer Analysis performance (in ms) of our approach compared to the estimate of brute force. Brute-force scales exponentially in terms of inference time, and our approach shows near constant scaling. The comparisons for DefUse and reducibility analyses are analogous. As with fact extraction, the results are most consistent for pointer analysis as the number of input facts is more consistent across files for that analysis. We note that there was no notable overhead for using our approach versus brute force on files with few features, but performance gains become very significant even in files with only 3 features. For output storage usage, the difference is scaling is the same as for input storage, discussed in the previous section.

5.2.4 Answer to RQ1 and RQ2

Evaluation results of the C-language benchmarks reinforce the answers to **RQ1** and **RQ2** demonstrated by the Java benchmarks. In addition, consistency of evaluations results across two different languages of benchmarks (Java and C), and two different variability annotation mechanisms (CPP and CIDE), allows us to answer **RQ3**. Neither the language of the benchmarks nor the annotation mechanism seem to affect performance gains both for fact extraction and inference.

5.3 Threats to Validity

For internal threats, we note that all of our Java benchmarks are CIDE product lines. While our lifting approach and implementation are not specific to CIDE (as manifested by our CPP-based evaluation experiments), CIDE limitations make the benchmarks biased towards specific annotation patterns. For example, only well-behaved annotations are allowed. Furthermore, since feature expressions do not support feature negation, all input PCs, as well as conjunctions over those PCs, are satisfiable. We experimented with disabling satisfiability checks (**RQ2.1**) to see how much they affect performance (while they always return `True` for this set of benchmarks). As noted previously, the overhead of those checks is marginal.

Another internal threat is that we approximate average product performance using only two samples (the maximum and the minimum). These averages are not expected

5. <https://souffle-lang.github.io/examples>

to be completely accurate, but are used to give a brute-force estimate. Our experiments show performance improvement of several orders of magnitude, so we believe that our approximation (compared to more elaborate configuration sampling techniques) can be tolerated.

Finally, our attempt to answer **RQ3** took only one step in the direction of validating the generality of results. All of the Java analyses we used come from the Doop framework, and all of our C-language analyses were hand-crafted, and applied to only one product line (BusyBox). The inference engine is independent of the fact extraction mechanism used, so facts can be extracted in a similar way from annotated product lines implemented in any programming language.

Mitigating the above threats to validity would involve targeting product lines implemented in more languages, and a wider range of analyses. Practically, the limiting factor here is the need to implement variability-aware fact extractors for each of those languages, which is not trivial. We plan to expand our evaluation to include additional Java and C-language analyses in the future.

6 RELATED WORK

Different kinds of software analyses have been re-implemented to support product lines [33]. For example, the TypeChef project [5], [6] implements variability aware parsers [5] and type checkers [6] for Java and C. The SuperC project [4] is another C language variability-aware parser. The Henshin [34] graph transformation engine was lifted to support product lines of graphs [8]. Those lifted analyses were written from scratch, without reusing any components from their respective product-level analyses. Our approach, on the other hand, lifts an entire class of product-level analyses written as Datalog rules, by lifting their inference engine (and extracting presence conditions together with facts). Our language-based lifting approach has been extended in [35] to a functional programming language.

SPL^{Lift} [2] extends IFDS [36] data flow analyses to product lines. Model checkers based on Featured Transition Systems [37] check temporal properties of transition systems where transitions can be labeled by presence conditions. Both of these SPL analyses use almost the same single-product analyses on a lifted data representation. At a high level, our approach is similar in the sense that the logic of the original analysis is preserved, and only data is augmented with presence conditions. Still, our approach is unique because we do not touch any of the Datalog rules comprising the analysis logic itself.

Syntactic transformation techniques have been suggested for lifting abstract interpretation analyses to SPLs [7]. This line of work outlines a systematic approach to lifting abstract interpretation analyses, together with correctness proofs. Yet this approach is not automated which means lifted analyses still need to be written from scratch, albeit while being guided by some systematic guidelines.

Datalog engines have been used as backends by several program analysis frameworks. In addition to Doop, examples of

analysis frameworks based on logic programming include XSB [22], bdbdbdb [38] and Paddle [10]. DIMPLE [21] is another declarative pointer analysis framework where rules are written in Prolog. To the best of our knowledge, all those program analysis frameworks have been targeting single products. Our primary contribution is lifting this class of analyses to SPLs in a generic way, without making any analysis-specific assumptions. In addition, our approach can be systematically implemented in any Datalog engine used by any of those frameworks.

7 CONCLUSION

In this paper, presented an algorithm for lifting Datalog-based software analyses to SPLs. We implemented this algorithm in the Soufflé Datalog engine, and evaluated performance of three program analyses from the Doop framework on a suite of Java and C-language SPL benchmarks. Comparing our lifted implementation to brute-force analysis of each product individually, we show significant savings in terms of processing time and database size.

Our Soufflé implementation only lifts the interpreter but not the code generator (compiler). Aggregation functions (e.g., sum, count) are not currently lifted either. We plan to address these implementation level limitations in future work. We also plan to evaluate lifted Soufflé on analyses frameworks other than Doop. Another track for future work is lifting Datalog rules, not just facts. This would allow us to apply a product line of analyses to an SPL all at once. For example, rules of the different flavors of pointer and taint analysis in the Doop framework can be annotated with presence conditions, and the entire set of Doop analyses (or a user-specified subset) can be applied to a single product or to a product line at once. This would add another dimension of performance savings because of the overlapping rules between the different analyses.

Our work can also be extended to lift analysis and verification tools based on Constrained Horn Clauses (CHC) [39] to support SPLs.

REFERENCES

- [1] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 81–91. [Online]. Available: <http://doi.acm.org/10.1145/2491411.2491437>
- [2] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini, "Spllift: Statically analyzing software product lines in minutes instead of years," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 355–364. [Online]. Available: <http://doi.acm.org/10.1145/2491956.2491976>
- [3] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: Efficient verification of temporal properties in software product lines," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 335–344. [Online]. Available: <http://doi.acm.org/10.1145/1806799.1806850>

- [4] P. Gazzillo and R. Grimm, "Superc: Parsing all of c by taming the preprocessor," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 323–334. [Online]. Available: <http://doi.acm.org/10.1145/2254064.2254103>
- [5] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 805–824. [Online]. Available: <http://doi.acm.org/10.1145/2048066.2048128>
- [6] C. Kästner, S. Apel, T. Thüm, and G. Saake, "Type checking annotation-based product lines," *ACM Trans. Eng. Methodol.*, vol. 21, no. 3, pp. 14:1–14:39, Jul. 2012. [Online]. Available: <http://doi.acm.org/10.1145/2211616.2211617>
- [7] J. Midtgaard, A. S. Dimovski, C. Brabrand, and A. Wasowski, "Systematic derivation of correct variability-aware program analyses," *Sci. Comput. Program.*, vol. 105, no. C, pp. 145–170, Jul. 2015. [Online]. Available: <http://dx.doi.org/10.1016/j.scico.2015.04.005>
- [8] R. Salay, M. Famelis, J. Rubin, A. Di Sandro, and M. Chechik, "Lifting model transformations to product lines," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 117–128. [Online]. Available: <http://doi.acm.org/10.1145/2568225.2568267>
- [9] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 422–430.
- [10] O. Lhoták and L. Hendren, "Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 1, pp. 3:1–3:53, Oct. 2008. [Online]. Available: <http://doi.acm.org/10.1145/1391984.1391987>
- [11] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 243–262. [Online]. Available: <http://doi.acm.org/10.1145/1640089.1640108>
- [12] R. Shahin, M. Chechik, and R. Salay, "Lifting datalog-based analyses to software product lines," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 39–49. [Online]. Available: <https://doi.org/10.1145/3338906.3338928>
- [13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 — Object-Oriented Programming*, M. Akšit and S. Mat-suoka, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242.
- [14] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.
- [15] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella, "Delta-oriented programming of software product lines," in *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, ser. SPLC'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 77–91.
- [16] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory, "Guaranteeing syntactic correctness for all product line variants: A language-independent approach," in *Objects, Components, Models and Patterns*, M. Oriol and B. Meyer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 175–194.
- [17] R. E. Lopez-Herrejon and D. S. Batory, "A standard problem for evaluating product-line methodologies," in *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, ser. GCSE '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 10–24.
- [18] M. Huth and M. Ryan, *Logic in Computer Science (2nd ed.)*. Cambridge University Press, 2004.
- [19] S. Greco and C. Molinaro, *Datalog and Logic Databases*. Morgan & Claypool, 2016.
- [20] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 146–166, March 1989.
- [21] W. C. Benton and C. N. Fischer, "Interactive, scalable, declarative program analysis: From prototype to implementation," in *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ser. PPDP '07. New York, NY, USA: ACM, 2007, pp. 13–24. [Online]. Available: <http://doi.acm.org/10.1145/1273920.1273923>
- [22] S. Dawson, C. R. Ramakrishnan, and D. S. Warren, "Practical program analysis using general purpose logic programming systems—a case study," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, ser. PLDI '96. New York, NY, USA: ACM, 1996, pp. 117–126. [Online]. Available: <http://doi.acm.org/10.1145/231379.231399>
- [23] N. Grech and Y. Smaragdakis, "P/taint: Unified points-to and taint analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 102:1–102:28, Oct. 2017. [Online]. Available: <http://doi.acm.org/10.1145/3133926>
- [24] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Foundations and Trends in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015. [Online]. Available: <http://dx.doi.org/10.1561/25000000014>
- [25] M. Madsen, M.-H. Yee, and O. Lhoták, "From datalog to flix: A declarative language for fixed points on lattices," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 194–208. [Online]. Available: <http://doi.acm.org/10.1145/2908080.2908096>
- [26] C. J. Date, *An Introduction to Database Systems (8th Edition)*. Pearson/Addison Wesley, 2004.
- [27] F. Somenzi, "Cudd: Cu decision diagram package release 2.2.0," 06 1998.
- [28] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba, "Intraprocedural dataflow analysis for software product lines," in *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, ser. AOSD '12. New York, NY, USA: ACM, 2012, pp. 13–24. [Online]. Available: <http://doi.acm.org/10.1145/2162049.2162052>
- [29] M. Sinn, F. Zuleger, and H. Veith, "A simple and scalable static analysis for bound analysis and amortized complexity analysis," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 745–761.
- [30] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975.
- [31] J. S. Pietro Abate, "Find all circuits of a directed graph using Johnson's algorithm and OCaml implementation by Pietro Abate," https://github.com/josch/cycles_johnson_abate, 2013.
- [32] M. Pezzè and M. Young, *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.
- [33] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 6:1–6:45, Jun. 2014. [Online]. Available: <http://doi.acm.org/10.1145/2580950>
- [34] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced concepts and tools for in-place emf model transformations," in *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I*, ser. MODELS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 121–135. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1926458.1926471>

- [35] R. Shahin and M. Chechik, "Automatic and efficient variability-aware lifting of functional programs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: <https://doi.org/10.1145/3428225>
- [36] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61. [Online]. Available: <http://doi.acm.org/10.1145/199448.199462>
- [37] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin, "Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1069–1089, Aug. 2013. [Online]. Available: <http://dx.doi.org/10.1109/TSE.2012.86>
- [38] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using datalog with binary decision diagrams for program analysis," in *Proceedings of the Third Asian Conference on Programming Languages and Systems*, ser. APLAS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 97–118. [Online]. Available: http://dx.doi.org/10.1007/11575467_8
- [39] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, *Horn Clause Solvers for Program Verification*. Cham: Springer International Publishing, 2015, pp. 24–51. [Online]. Available: https://doi.org/10.1007/978-3-319-23534-9_2