Annotative Software Product Line Analysis Using Variability-Aware Datalog

Ramy Shahin®, Murad Akhundov®, and Marsha Chechik

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 annotative software 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. In particular, annotative SPLs use artifact annotation techniques (e.g., the C Pre-Processor (CPP)) to define which set of product configurations each artifact belongs to. The number of potential products is typically combinatorial in the number of features because each feature can be present or absent in each product configuration. This high level of configurability is usually desired. For example, the open-source Linux kernel is an annotative product line of more than 10,000 features [1]. Different variants of the kernel can be generated at build-time by providing different feature combinations (subject to feature dependencies). However, analysis tools (syntax analyzers, type checkers, model checkers, static analysis tools, etc...) typically work on a single product, not the whole annotative SPL. Applying an analysis to each product separately is usually infeasible for non-trivial SPLs because of the exponential number of products [2].

Analysis tools are valuable during the initial implementation of a product line, and they are even more valuable detecting modifications that break existing functionality during product line maintenance. Since all products of an annotative SPL share a common set of artifacts, analyzing each product individually (usually referred to as brute-force analysis) would involve a lot of redundancy. Leveraging this commonality and analyzing the whole product line at once in order to bring down the total analysis time 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 [3], [4], [5], [6], [7], [8], [9]. Those attempts show significant time savings when the annotative SPL is analyzed as a whole compared to the brute-force analysis. The downside of lifting individual analyses 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 [10], [11].

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 annotative 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 [12] to extract facts from annotative 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 Busy-Box 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 our ESEC/FSE conference paper [13]. In particular: (1) We evaluate our approach on a wider range of product line benchmarks written in Java and C-language, and using different feature annotation mechanisms; (2) We discuss and illustrate the distinction between annotation mechanisms (C PreProcessor (CPP) and CIDE), with emphasis on its impact on variability-aware fact extraction; (3) We discuss variability-aware fact extraction in more detail; (4) We add more elaborate examples and illustrations throughout the paper.

The rest of the paper is organized as follows: We start with a background on SPLs and Datalog (Section 2). We provide a theoretical treatment of Datalog inference, how the inference algorithm is lifted, together with correctness criteria and a correctness proof in Section 3. In Section 4, we describe the implementation of our algorithm in the Soufflé engine. Evaluation process and results are discussed in Section 5. We compare our approach to related work in Section 6 and conclude (Section 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 \( (F, \Phi, D, \phi) \) where: (1) \( F \) is the set of features s.t. an individual product can be derived from \( \mathcal{L} \) via a feature configuration \( \rho \subseteq F \). A feature configuration can be also represented as a propositional formula over \( F \), conjointing the features included in the configuration with the negation of the features excluded. (2) \( \Phi \in \text{Prop}(F) \) is a propositional formula over \( F \) defining the valid set of feature configurations. \( \Phi \) is called a Feature Model (FM). The set of valid configurations defined by \( \Phi \) is called 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 [14]. (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 [15] and Feature-Oriented Programming [16]. Delta-oriented SPLs [17] 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 [18] 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, code in line 10 is in the scope of annotation \( \neg FA \). That is, \( \neg FA \) is the Presence Condition (PC) of these tokens. Similarly, tokens on line 13 have the PC \( FB \). This SPL allows all four feature combinations, so its feature model \( \Phi \) is the propositional formula True.

Listing 1. A product line with features \( FA \) and \( FB \).

```java
1   class Parent {
2     public Object f;
3   }
4
5   class ClassA extends Parent {
6     new ClassA();
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 2. A product with configuration \( \{ FB \} \) (can be also represented as \( \neg FA \land FB \)) [19].

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

CIDE Example. The code snippet in Fig. 1 comes from the Graph Product Line (GPL) benchmark [20], with featurespecific code highlighted in different colors. In this example, the light-blue color denotes the EdgeObjects feature, while the violet color denotes the feature combination EdgeObjects \( \land \) Weighted.
CIDE only allows well-behaved annotations, i.e., annotations enforcing the syntactic correctness and type-checking of each variant. 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).

2.2 Horn Clauses and Datalog

2.2.1 Horn Clauses

A Horn Clause (HC) is a disjunction of distinct propositional literals with at most one positive literal. For example, \((\neg a \lor \neg b \lor c \lor \neg d)\) is an HC, which can be also written as a reverse-implication \((c \leftarrow (a \land b \land d))\), where \(c\) is called the head and \((a \land b \land d)\) is called the body of the clause. The language of HCs is a fragment of Propositional Logic that can be checked for satisfiability in time linear in the number of input facts, as opposed to general propositional satisfiability which is NP-complete [21].

2.2.2 Datalog

Datalog is a declarative database query language that extends relational algebra with logical inference [22]. 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 \(\text{Parent}\), a binary predicate \(\text{Ancestor}\) can be defined using the following two rules:

\[
\text{Ancestor}(x, y) :\leftarrow \text{Parent}(x, y).
\]

\[
\text{Ancestor}(x, z) :\leftarrow \text{Parent}(x, y), \text{Ancestor}(y, z).
\]

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, where each variable appearing in the head also appears in the body of the clause; 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 more new facts can be inferred.

Algorithm 1. Inference Algorithm infer (Forward Chaining)

\[
\begin{align*}
1 & \text{repeat} \\
2 & \text{fixpoint } \leftarrow \text{True}; \\
3 & \text{foreach } (C \leftarrow L_1, \ldots, L_n) \in \text{IDB} \text{ do} \\
4 & \text{foreach } (\gamma; F_1, \ldots, F_m), F_i \in \text{EDB}, [\gamma]L_i = F_i \text{ do} \\
5 & \quad F_c \leftarrow [\gamma]C; \\
6 & \quad \text{if } F_c \notin \text{EDB} \text{ then} \\
7 & \quad \text{fixpoint } \leftarrow \text{False}; \\
8 & \quad \text{EDB } \leftarrow \text{EDB } \cup \{F_c\}; \\
9 & \quad \text{end} \\
10 & \quad \text{end} \\
11 & \quad \text{end} \\
12 & \text{until fixpoint} \\
13 & \text{return EDB};
\end{align*}
\]

2.3 Example of a Datalog Analysis
Some program analyses [12], [25], [26], [27] 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.

\[
\begin{align*}
\text{New}("o1", "A"). & \quad \text{// line 5} \\
\text{New}("o2", "B"). & \quad \text{// line 6} \\
\text{Assign}("o3", "o2"). & \quad \text{// line 7} \\
\text{Store}("o2", "f", "o1"). & \quad \text{// line 8} \\
\text{Load}("r", "o3", "o1"). & \quad \text{// line 9}
\end{align*}
\]

(b) Facts extracted from Listing 2 (EDB).

\[
\begin{align*}
\text{VarPointsTo}("o1", "A"). & \\
\text{VarPointsTo}("o2", "B"). & \\
\text{VarPointsTo}("o3", "B"). & \\
\text{HeapPointsTo}("B", "f", "A"). & \\
\text{VarPointsTo}("r", "A"). &
\end{align*}
\]

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

Pointer analysis [28] 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. 4 shows a set of Datalog rules for a simple context-insensitive pointer analysis [19]. Each predicate defines a relation between different artifacts. \text{VarPointsTo}(v, h) states that pointer \( v \) might point to a heap object \( h \). \text{New}(v, h) means an allocation of...
a heap object $h$ is assigned to a pointer $v$. Assign($u$, $v$) captures an assignment from a pointer $v$ to a pointer $u$. For object fields, HeapPointsTo($o$, $f$, $h$) means that a field $f$ of an object $o$ might point to a heap object $h$. Load($u$, $v$, $f$) represents a load of a field $f$ of an object pointed to by $v$ into a pointer $u$. Similarly, Store($u$, $v$, $f$) represents a store of the object pointed to by $v$ to a field $f$ of the object pointed to by $u$.

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 [28]. 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. Fig. 3c presents the grammar of Datalog clauses, annotated with presence conditions over a set of features.

Facts annotated with PCs are called lifted facts, and are stored in a lifted Extended Database – $\text{EDB}_\rho$. Given a feature configuration $\rho$, we define $\text{EDB}_\rho$ (restriction) to be the set of facts from $\text{EDB}$ which only exist in the product configuration defined by $\rho$

$$\text{EDB}_\rho = \{ f \mid (f, pc) \in \text{EDB} \land \text{sat}(pc \land \rho) \}.$$  

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. The core of the lifted Datalog inference (Algorithm 2) is the lifted Modus Ponens inference rule ($MP$) in Fig. 5. If $C$ is generated from premises $F_1, F_2, \ldots, F_n$, with PCs $pc_1, \ldots, pc_n$, then $pc_{c}$ attached to $C$ should be the conjunction of the input PCs, i.e., $pc_1 \land \ldots \land pc_n$. Intuitively, $pc_{c}$ represents the set of products in which $C$ 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 $pc_{c}$ 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 $\text{EDB}$, optimizing the number of facts stored in the database, and potentially improving the performance of inference. There is a trade-off though between the computational complexity of satisfiability checking (an NP-complete problem), and optimizing the database size. We address this trade-off in more detail in Section 5.1.3.

3.2 Lifted Inference Algorithm

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 according to $MP$ (Fig. 5), and assigning the conjunction as the presence condition of the result. There are four cases for $(c, pc_{c})$ to consider. (1) if 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 $(c, pc_{c}) \in \text{EDB}$, then this result is also ignored because it already exists for the same set of products; (3) if $(c, pc_{c}) \in \text{EDB}$, where $(pc_{c} \lor pc_{d})$ is not logically equivalent to $pc_{d}$, then $(c, pc_{d})$ is replaced with $(c, pc_{c} \lor pc_{d})$ in $\text{EDB}$. This means we are expanding the already existing set of products in which $c$ exists to also include the set denoted by $pc_{c}$. This accounts for the case where the products denoted by $pc_{c}$ are a subset of the products denoted by $pc_{d}$. In such cases, the disjunction of $pc_{c}$ and $pc_{d}$ is equal to $pc_{d}$, and the existing set of products is not expanded. Avoiding storing the same fact multiple times in the database, each time with a different presence condition, optimizes the database size, at the expense of the disjunction cost of those presence conditions. It also makes the addition of lifted aggregation functions (e.g., sum, count) more straightforward in the future; (4) if $c$ doesn’t exist at all in $\text{EDB}$, 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. 6, the result is the following:

\[
\text{VarPointsTo}(*01, *A) \quad \text{VarPointsTo}(*02, *B) \quad \text{VarPointsTo}(*03, *A) \quad \text{VarPointsTo}(*03, *B) \quad \text{HeapPointsTo}(*B, *f, *A) \quad \text{HeapPointsTo}(*B, *f, *B) \quad \text{VarPointsTo}(*r, *A) \quad \text{VarPointsTo}(*r, *B) \quad \text{HeapPointsTo}(*B, *f, *B) \quad \text{VarPointsTo}(*r, *A) \quad \text{VarPointsTo}(*r, *B)
\]

The last two facts in the results show that the pointer \( r \) might be pointing to the object A when the feature FA is present, while it might be pointing to the object B if FA is absent.

**Algorithm 2. Lifted Inference Algorithm infer (Differences w.r.t. Algorithm 1 are in Green)**

**Input:** IDB, EDB

**Result:** EDB \cup \{annotated inferred clauses\}

1. repeat
2. 1. fixpoint \leftarrow True;
3. 2. foreach \((C \Rightarrow L_1, \ldots, L_n) \in IDB\) do
4. 3. foreach \((y, (F_i, pc_i), \ldots, (F_n, pc_n)), (F_i, pc_i) \in EDB, [y]L_i = F_i\) do
5. 4. \(F_i \leftarrow [y]C;\)
6. 5. \(pc_c \leftarrow pc_1 \wedge \ldots \wedge pc_n;\)
7. 6. if \(sat(pc_c)\) then
8. 7. if \(3pc_c, (F_c, pc_c) \in EDB\) then
9. 8. if \(pc_c \vee pc_d = pc_c\) then
10. 9. fixpoint \leftarrow False;
11. 10. EDB \leftarrow EDB \setminus \{(F_c, pc_c)\};
12. 11. EDB \leftarrow EDB \cup \{(F_c, pc_c \vee pc_d)\};
13. 12. end
14. 13. else
15. 14. fixpoint \leftarrow False;
16. 15. EDB \leftarrow EDB \cup \{(F_c, pc_c)\};
17. 16. end
18. 17. end
19. 18. end
20. 19. end
21. until fixpoint
22. return EDB;

**3.3 Correctness Criteria**

When applying the lifted inference algorithm \(\text{infer}\) to a set of rules IDB and a set of annotated facts 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 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 \(\text{EDB}\) annotated with feature expressions over \(F\):

\[
\forall (\rho \in \text{Conf}(\mathcal{L})), \overline{\text{infer}(\text{EDB})}_\rho = \text{infer}(\text{EDB|}_\rho).
\]

**Proof.** The proof breaks the equality into two implications of set membership (forward and backward). Both subproofs rely on the fact that infer is monotonic in the sense that new facts are added to \(\text{EDB}\) without removing any of the existing facts. The logic behind inferring a new fact is summarized by the MP inference rule (Fig. 5). As stated in Algorithm 2 (lines 11-12), when an element is removed from \(\text{EDB}\), it is replaced by the same fact with an extended set of products. Also the algorithm is guaranteed to terminate because in each iteration either a new fact is added to the fact base, or the set of products of an existing fact is expanded. Because new facts do not introduce new symbols, this procedure will always reach a fixed point.

**Forward Implication.**

\[
C \in \overline{\text{infer}(\text{EDB})}_\rho \Rightarrow C \in \text{infer}(\text{EDB|}_\rho).
\]

By structural induction over the derivation tree of \(C\): 

**Base Case:** The base case is a clause \(C\) that already exists in the input of the lifted inference algorithm. In other words, for some presence condition \(pc, (C, pc) \in \text{EDB}\). Since \(C \in \text{infer}(\text{EDB|}_\rho)\) then \(sat(pc \wedge \rho)\) and, subsequently, \(C \in \text{EDB|}_\rho\) (by definition of restriction operator). Since inputs are already included in the output of infer, \(C \in \text{infer}(\text{EDB|}_\rho)\).

**Induction Hypothesis:** For each rule \(R = C : \ldots, L_n\) in the IDB and a variable assignment \(\gamma\),

\[
\forall_{1 \leq i \leq n} \gamma[L_i \in \text{infer}(\text{EDB|}_\rho) \Rightarrow \gamma[L_i \in \text{infer}(\text{EDB|}_\rho)]
\]

**Induction Step:** Since \(C \in \overline{\text{infer}(\text{EDB})}_\rho\), there is a presence condition \(pc\) where \((C, pc) \in \text{infer}(\text{EDB})\) and \(sat(pc \wedge \rho)\) (definition of restriction operator). Because \(C\) is derived using an \(n\)-ary rule \(R \in \text{IDB}\), where the unification of each of the premises \(L_i\) of \(R\) using variable assignment \(\gamma\) exists in \(\text{EDB}\) with a presence condition \(pc_\circ\), then \(pc_\circ \wedge pc_i\) (definition of infer). Since \(pc \wedge \rho\) is satisfiable, then \(sat(pc \wedge \rho)\) (each of the conjuncts has to be satisfiable). Then \(\forall_{1 \leq i \leq n} \gamma[L_i \in \text{infer}(\text{EDB|}_\rho)]\). By induction hypothesis, \(\forall_{1 \leq i \leq n} \gamma[L_i \in \text{infer}(\text{EDB|}_\rho)]\). By definition of infer, \(C \in \text{infer}(\text{EDB|}_\rho)\).

**Backward Implication.**

\[
C \in \text{infer}(\text{EDB|}_\rho) \Rightarrow C \in \overline{\text{infer}(\text{EDB})}_\rho.
\]

By structural induction over the derivation tree of \(C\):

**Base Case:** The base case is a clause \(C\) that already exists in the input of the lifted inference algorithm. In other words, for some presence condition \(pc, (C, pc) \in \text{EDB}\). Since \(C \in \text{infer}(\text{EDB|}_\rho)\), \(pc \wedge \rho\) is satisfiable (definition of restriction operator). Then \(C \in \text{infer}(\text{EDB|}_\rho)\).

**Induction Hypothesis:** For each rule \(R = C : \ldots, L_n\) in the IDB and a variable assignment \(\gamma\),

\[
\forall_{1 \leq i \leq n} \gamma[L_i \in \text{infer}(\text{EDB|}_\rho) \Rightarrow \gamma[L_i \in \text{infer}(\text{EDB|}_\rho)]
\]
Induction Step: Since $C \in \text{infer}(\text{EDB}|p)$, then $C$ is derived using an $n$-ary rule $R \in \text{IDB}$, where the unification of each of the premises $L_i$ of $R$ using variable assignment $\gamma$ exist in $\text{infer}(\text{EDB}|p)$. Then the unification of each of those premises using the variable assignment $\gamma$ also exists in $\text{infer}(\text{EDB}|p)$ (induction hypothesis). For each premise $([\gamma]L_i, pc_i)$, $pc_i \land \rho$ is satisfiable (definition of restriction operator). Since $\rho$ defines a single product instance, then for each of the premises, $pc_i \land \rho = \rho$. The conjunction of the presence conditions of the premises together with $\rho$ equals $\rho$. Then $(C, \rho) \in \text{infer}(\text{EDB})$ (definition of infer), and $C \in \text{infer}(\text{EDB})|\rho$ (definition of restriction operator).

Note that the way we expand the set of products associated with a fact (line 10, Algorithm 2) preserves the correctness guarantees. The rationale is if a fact holds for a set of products $s_1$ represented by $pc_1$, and another set $s_2$ represented by $pc_2$, then intuitively it holds for the union of both sets ($s_1 \cup s_2$), represented by $pc_1 \lor pc_2$.

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 [29] to make it variability-aware (Section 4.1). Several other Datalog engines are used as backends for program analysis frameworks (e.g., LogicBlox [24]). We chose to lift Soufflé because it implements several optimizations that make it scalable for analyzing relatively big systems, it is available in open source, and it serves as the backend for the Doop pointer analysis framework [12] that we use in our evaluation. Doop implements several pointer and taint analyses of Java programs, where each analysis is represented as a set of Datalog rules. Doop extracts syntactic facts from Java bytecode and passes those facts and the Datalog rules of the particular analysis chosen by the user to a backend Datalog engine (the default backend is Soufflé). The results of the analysis are then generated by the Datalog engine. We outline the modifications we made to the Doop fact extractor (extracting facts from Java bytecode) to make it variability-aware (Section 4.2.1). Finally, we outline the design of a C-language fact extractor, and a set of Datalog analyses for C programs (Section 4.2.2).

The overall architecture of the two analysis pipelines is outlined in Fig. 7. The Doop pipeline takes “Java Bytecode” files (150% representations of product lines annotated with CIDE) 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 (150% representations of product lines annotated with the C Pre-Processor (CPP)) as input. Because variability in those files is annotated using the 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 [10] is a highly optimized open source Datalog engine. Our modified variability-aware version of Soufflé (V-Soufflé) implements our lifted Datalog inference algorithm infer (Section 3).

As seen in Fig. 8, 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 [30], 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++
At the syntax level, we – prop - which uses Souffl. The RAM Interpreter takes a RAM program as input, and executes it on the fly. This involves both the Datalog forward-chaining inference algorithm (infer) to infer new facts, and applying relational algebra operators such as selection, projection and joins. Souffl 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 Souffl 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 to be able to support a wider range of analyses, particularly those that aggregate results over multiple facts.

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 Section 3, we modified the Doop [12] Datalog-based pointer analysis framework,2 which uses Souffl as its underlying Datalog engine. Fig. 7 shows the Doop architecture together the rest of the analysis pipeline. Doop is an 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 Souffl.

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 (Section 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. 7 for the high-level architecture. Unlike our Java fact extractor, we designed our C fact extractor from scratch. Existing frameworks often

2. Available online at https://bitbucket.org/rshahin/doop
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;
}

Fig. 9. A simple C program with variability.

used for C program analysis such as CIL\(^3\) and LLVM\(^4\) only work with pre-processed code, with no support for variability. Performing analysis directly on C instead of LLVM IR or CIL is significantly more difficult due to the more complex syntax and semantics of C. We describe it below.

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\(^6\) 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. 10 shows the V-CFG generated by TypeChef for the program in Fig. 9. The format of the file for nodes consists of the lines starting with the character “N,” followed by the identifier of the node, function name, 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, 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. Each `PresenceConditionNode` in the V-AST corresponds to an `ifdef` from the program. Fig. 12 shows the facts generated from the V-AST in Fig. 10 where the FactDef functions identify and extract facts for pointer analysis, as described in Section 5.2.

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 generate a V-CFG with TypeChef – this process automatically splits statements that contain variability – and then parse ASTs of those statements.

Algorithm 3. Generating Facts From V-AST. FactDef is a Pair of Functions `fits()` and `gets()`. 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. CurrCond Keeps Track of the Current Presence Condition. For Non-Recursive Calls, CurrCond is Set to True.

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

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 function returns a list of desired relationships, and a list of nodes related to each relationship. We used OCamlgraph\(^5\) to assist with control-flow analysis, specifically, with identifying node domination relationships. The `cfgAnalysis` procedure

3. https://github.com/cil-project/cil
4. https://llvm.org/
5. http://ocamlgraph.lri.fr/
can be a wrapper around OCamlGraph or a different
directed graph analysis utility. Fig. 13 shows simple domina-
tor relationship facts extracted from the program in Fig. 9,
using Algorithm 4. The first and second parameters repre-
sent the dominating and the dominated nodes, respectively,
and cfgAnalysis() is a function to find all dominator pairs in
a directed graph.

We assume that the V-CFG generated by TypeChef is
correct and that cfgAnalysis() works as described. Our cor-
rectness criterion is that the presence condition of the
extracted fact is True if and only if the presence condi-
tions 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.

Algorithm 4. Generating Facts From V-CFG. The Algo-
rithm Uses a Helper Function cfgAnalysis() That Performs
the Desired Directed Graph Analysis Returning Pairs of
Extracted Facts and the Nodes Relating to Them.

```
Input: V-CFG, cfgAnalysis
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 ← \( \land \) (conditions)
8   facts.append((fact, presenceCond))
9 end
```

5 EVALUATION

We evaluate the performance of our lifted version of Soufflè
using two sets of benchmarks: Three Doop analyses applied
to five Java benchmark product lines (previously used in
the evaluation of other lifted analyses \[3\], \[32\]), and the
three hand-crafted C-language analyses outlined in Sec-
tion 5.2 applied to the BusyBox \(^6\) 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 rel-
atively 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 var-
nants, 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:

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 Soufflè 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 frame-
work. 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.

| Benchmark | Size (KLOC) | Features | Configs |
|-----------|-------------|----------|---------|
| BusyBox   | 1332        |          |         |

Fig. 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.

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

```
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
```

6. https://www.busybox.net/
details on each of the benchmarks can be found in the SPL2go\textsuperscript{7} catalog of software product lines. For example, BerkeleyDB is about 70,000 lines of code, comprised of 42 features, and has about 8.76 billion valid product configurations. Given a propositional representation of the feature model of each of the benchmarks, we calculated the number of valid configurations using a SAT\# solver (counting the number of distinct models satisfying the feature model).

We use three analyses from the Doop framework in our evaluation of the Java benchmarks:

\textbf{Context-Insensitive Pointer Analysis} (insens). This is the simplest (and least-precise) pointer analysis in the Doop framework. It does not take any context (e.g., object context, call context) into consideration when trying to resolve which heap objects each pointer might be pointing at.

\textbf{One-Type Heap-Sensitive Pointer Analysis} (1Type+Heap). This is a pointer analysis that takes the type (Java class) in which a heap allocation takes place into consideration. Allocations taking place in the same Java class are merged together.

\textbf{One-Call-Site Heap-Sensitive Taint Analysis} (Taint-1Call+Heap). This is a taint analysis that takes the context of the direct caller of a function (but not callers of callers) into consideration [28]. We use the default sources, sinks, transform and sanitization functions curated in Doop for the JDK and Android [27].

All experiments were performed on a Quad-core Intel Core i7-6700 processor running at 3.4 GHZ, with 16 GB 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 \textit{disciplined annotations} [18], 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.

The benchmarks we chose are annotated using CIDE [18], 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.

\begin{table}[h]
\centering
\caption{Java Product Lines Used for Evaluation}
\begin{tabular}{llll}
Benchmark & Size (KLOC) & Features & Valid Configurations \\
\hline
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 \\
\end{tabular}
\end{table}

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 1,416 ms, and the average size of the extracted fact database is 3,230 KB. On the other hand, extracting facts from the whole Prevayler SPL at once takes 1,554 ms, and the extracted fact database is 4,407 KB. 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-Avg / SPL, P-Avg * |Conf(L)| / 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 \textit{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 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 \textit{Taint-1Call+Heap} databases are significantly bigger than those of \textit{1Type+Heap}. BerkeleyDB, on the other hand, exhibits the opposite trend.

\begin{table}[h]
\centering
\caption{Java Product Lines Used for Evaluation}
\begin{tabular}{llll}
Benchmark & Size (KLOC) & Features & Valid Configurations \\
\hline
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 \\
\end{tabular}
\end{table}

7. https://spl2go.cs.ovgu.de/
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, when running 1Type+Heap on an average MM08 product, inference is estimated to take 5,106 ms resulting in a 19,058 KB database. Running the same analysis on the entire MM08 product line, though, takes 5,142 ms, resulting in a 29,453 KB database. Fig. 15 is a log-scale bar graph of the speedup factor and the DB size 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 theoretical bottleneck of the lifted inference algorithm (Algorithm 2) is the satisfiability checks performed when conjoining two PCs (Section 3.1). 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 4 summarizes the performance of our lifted analyses and the same analyses with SAT checking disabled (noSAT). Figs. 16 and 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 [21]. 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 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 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. Figs. 18 and 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 evaluated our C-language analyses on 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 in every node of the V-CFG generated by TypeChef. The resulting 229 files were used in our evaluation.

We have implemented three Datalog analyses which we describe below.8

**Context-Insensitive Pointer Analysis.** Similar to context-insensitive pointer analysis in Doop, the goal is to statically identify the set of heap objects to which each pointer might be pointing. Extracted facts include variable assignments and functions returning pointers. Since 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. Fig. 12 shows an example of input facts used by this analysis, with assign facts representing assignment statements and allocHeap facts representing malloc statements. The analysis generates pointsTo facts indicating a mapping between the variables and the allocated heap space.

**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 [33]. This analysis identifies all possible cycles in the graph using Johnson’s algorithm [34]. We used Pietro Abate’s functional OCaml implementation [35] 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 the 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 [36]. We used a Datalog implementation of this analysis from the Soufflé tutorials.9 Extracted facts for this analysis include all edges in the CFG, variable loads (references), and variable stores (assignments). Each load and store fact contains both the identifier of the relevant CFG node and the variable being stored or loaded. The facts for edges correspond to the entire V-CFG generated by TypeChef for that program. The inference algorithm computes and outputs def-use CFG node pairs using the rules of this analysis.

We compared our lifted approach against the estimate of brute-force analysis – see Section 5.1. Fig. 22 represents the distribution of features-per-file for the files we analyzed. The majority of the files had between zero and two features and thus we expected the performance of our analysis to be similar to our brute force estimate. 90 files had three or more features, and thus we expected to see our approach yield a more substantial improvement in these cases. We performed the experiments on a server with a 3 GHz 32-core 64-bit processor, 128 GB of RAM, running Ubuntu Linux 18.04.

5.2.2 Fact Extraction

Fig. 21b shows pointer analysis performance, in ms, of our approach compared to the estimate of brute force. Our approach demonstrated clear, near-constant growth in terms of fact extraction time, while brute force demonstrated exponential growth. The difference between our approach and the estimate of brute force is similar in def-use and reducibility analyses. However, these analyses result in larger differences between files with the same number of features. This is because these analyses rely on the CFG structure and relationships between CFG nodes; these can vary vastly depending on the structure and the size of the file, resulting in much

---

8. Available online at https://github.com/MuradAkh/Variability-Datalog-C
9. https://souffle-lang.github.io/examples
larger running times for some files independent of the number of features. We can observe the same pattern for the size of the extracted facts database across all analyses, as shown in Fig. 20. For both storage usage and performance, our approach showed most significant gains versus brute force in files with four or more features. Since fact extraction is a relatively cheap process, the gains in files with fewer than four features are often marginal.

5.2.3 Inference

Fig. 21b shows the running time of pointer analysis of our approach compared to the estimate of brute force (in ms). The improvement is comparable to that of fact extraction discussed in Section 5.2.2. The benefit in terms of size of the final inference output fact database is similar to that of the input fact database. We can observe a significant improvement in both performance and storage usage using our approach in files with as few as two-three features, depending on the analysis. Inference is generally significantly more expensive than fact extraction. Pointer analysis is an exception to this as the number of generated pointsTo facts usually grows linear to the number of assign and other extracted facts.

5.3 Answers to RQ1, RQ2, and RQ3

Evaluation results of the C-language benchmarks reinforce those of the Java benchmarks. For both sets of benchmarks, savings in fact extraction time and extracted fact database size grow exponentially with the number of configurations (RQ2). In addition, consistency of evaluation results across the two language benchmarks (Java and C), and the 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 for fact extraction or for the inference.

5.4 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 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.

### Table 4: Doop SAT versus 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 |
| GPL      | 4,517 | 4,496 | 409  | 472   | 4,718 | 4,667  | 593  | 765   |
| Lampiro  | 8,111 | 8,105 | 41,170 | 41,170 | 20,725 | 21,224 | 149,686 | 149,710 |
| MM08     | 4,720 | 4,689 | 3,259 | 3,762 | 5,142 | 5,118  | 6,990 | 8,775 |
| Prevalier | 5,334 | 5,050 | 4,407 | 4,638 | 6,013 | 5,940  | 8,630 | 9,392 |

Time in milliseconds, Inferred DB in KB.

Fig. 16. Doop inference speedup factor : SPL versus SPL with SAT checking disabled for all three analyses.

Fig. 17. Doop inferred database savings factor : SPL versus SPL with SAT checking disabled for all three analyses.
We did not measure the amount of effort we put in making fact extractors variability-aware. We assume that at least a product-level fact extractor (like the Doop fact extractor) already exists; otherwise, product-level analyses would not be usable. Variability-aware modifications to the product-level fact extractor depend on the programming language, the variability annotation mechanism, and the kind of facts to be extracted.

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 annotative 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 [37]. For example, the TypeChef project [6], [7] implements variability aware parsers [6] and type checkers [7] for Java and C. The SuperC project [5] is another C language variability-aware parser. The Henshin [38] graph transformation engine was lifted to support product lines of graphs [9]. 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
A language-based lifting approach has been extended in [39] to a functional programming language.

SPL Lift [3] extends IFDS [40] data flow analyses to product lines. Model checkers based on Featured Transition Systems [41] 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 [8]. 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

---

Fig. 21. C benchmarks inference and extraction time.
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 [26], bddbddb [42] and Paddle [11]. DIPLE [25] 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, we presented an algorithm for lifting Datalog-based software analyses to annotative 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) [43] to support SPLs.

References
[1] S. Nadi and R. Holt, “The Linux kernel: A case study of build system variability,” J. Softw., Evol. Process, vol. 26, no. 8, pp. 730–746, 2014.
[2] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lenganuer, “Scalable analysis of variable software,” in Proc. 9th Joint Meeting Found. Softw. Eng., 2013, pp. 81–91.
[3] E. Bodden, T. Toledo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini, “SPLIT: Statically analyzing software product lines in minutes instead of years,” in Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation, 2013, pp. 355–364.
[4] 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 Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng., 2010, pp. 335–344.
[5] P. Gazzillo and R. Grimm, “SuperC: Fasing all of C by taming the preprocessor,” in Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation, 2012, pp. 323–334.
[6] 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 Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl., 2011, pp. 805–824.
[7] C. Kästner, S. Apel, T. Thüm, and G. Saake, “Type checking annotation-based product lines,” ACM Trans. Softw. Eng. Methodol., vol. 19, no. 3, pp. 323–334.
[8] J. Midtgad, Å. S. Dimovski, C. Brabrand, and A. Wasowsk, “Systematic derivation of correct variability-aware program analyses,” Sci. Comput. Prog., vol. 105, no. C, pp. 145–170, Jul. 2015.
[9] R. Salay, M. Famelis, J. Rubin, A. Di Sandro, and M. Chechik, “Lifting model transformations to product lines,” in Proc. ACM/IEEE Int. Conf. Softw. Eng., 2014, pp. 117–128.
[10] H. Jordan, B. Schach, and P. Subotić, “Soufflé: On synthesis of program analyzers,” in Proc. Int. Conf. Comput. Aided Verification, 2016, pp. 422–430.
[11] 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, 2008, Art. no. 3.
[12] M. Bravenboer and Y. Smaragdakis, “Strictly declarative specification of sophisticated points-to analyses,” in Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl., 2009, pp. 243–262.
[13] R. Shahin, M. Chechik, and R. Salay, “Lifting datalog-based analyses to software product lines,” in Proc. Joint Meeting Found. Softw. Eng., 2019, pp. 39–49.
[14] D. Beuche, M. Schulze, and M. Duvigneau, “When 150% is too much: Supporting product centric viewpoints in an industrial product line,” in Proc. 20th Int. Symp. Softw. Product Line Conf., 2016, pp. 262–269.
[15] G. Kiczales et al., Aspect-oriented programming,” in Proc. Eur. Conf. Object- Oriented Program., 1997, pp. 220–242.
[16] S. Apel, D. Batory, C. Kstner, and G. Saake, Feature-Oriented Software Product Lines: Concepts and Implementation, Berlin, Germany: Springer Publishing Company, Incorporated, 2013.
[17] I. Schaefer, L. Bettini, F. Damiani, and N. Tzanzarella, “Delta-oriented programming of software product Lines,” in Proc. Int. Conf. Softw. Prod. Lines, 2017, pp. 72–77.
[18] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory, “Guaranteeing syntactic correctness for all product line variants: A language-independent approach,” in Proc. Int. Conf. Objects, Components, Models Patterns, 2009, pp. 175–194.
[19] M. Madsen, M.-H. Yee, and V. Lhotak, “From datalog to flx: A declarative language for fixed points on lattices,” in Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation, 2016, pp. 194–208.
[20] R. E. Lopez-Herrejon and D. S. Batory, “A standard problem for evaluating product-line methodologies,” in Proc. 3rd Int. Conf. Generative Component-Based Softw. Eng., 2001, pp. 10–24.
[21] M. Huth and M. Ryan, Logic in Computer Science (2nd ed.), Cambridge, U.K.: Cambridge Univ. Press, 2004.
[22] S. Greco and C. Molinaro, Datalog and Logic Databases. San Rafael, CA, USA: Morgan & Claypool, 2016.
[23] S. Ceri, G. Gottlob, and L. Tanca, “What you always wanted to know about datalog (and never dared to ask),” IEEE Trans. Knowl. Data Eng., vol. 1, no. 1, pp. 146–166, Mar. 1989.
[24] M. Arel et al., “Design and implementation of the logibox system,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2015, pp. 1371–1382.
[25] W. C. Benton and C. N. Fischer, “Interactive, scalable, declarative program analysis: From prototype to implementation,” in Proc. 9th ACM SIGPLAN Conf. Int. Proc. Pract. Declarative Program., 2007, pp. 13–24.
[26] S. Dawson, C. R. Ramakrishnan, and D. S. Warren, “Practical program analysis using general purpose logic programming systems – A case study,” in Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation, 1996, pp. 117–126.
[27] 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.
[28] Y. Smaragdakis and G. Balatsouras, “Pointer analysis,” Found. Trends Program. Lang., vol. 2, no. 1, pp. 1–69, 2015.
[29] R. Shahin and M. Chechik, “Variability-aware datalog,” in Proc. Int. Symp. Practical Aspects Declarative Lang., 2020, pp. 213–221.
[30] C. J. Date, An Introduction to Database Systems (8th ed.), London, U.K.: Pearson/Addison Wesley, 2004.
[31] F. Somensi, “CUDD: CU decision diagram package release 2.2.0,” Jun. 1998.
[32] C. Brabrand, M. Ribeiro, T. Toledo, and P. Borba, “Intraprocedural dataflow analysis for software product lines,” in Proc. 11th Ann. Int. Conf. Aspect-Oriented Softw. Develop., 2012, pp. 13–24.
[33] M. Sinn, F. Zuleger, and H. Veith, “A simple and scalable static analysis for bound analysis and amortized complexity analysis,” in Proc. Int. Conf. Comput. Aided Verification, 2014, pp. 745–761.
[34] D. B. Johnson, “Finding all the elementary circuits of a directed graph,” SIAM J. Comput., vol. 4, no. 1, pp. 77–84, 1975.
[35] J. S. Pietro Abate, “Find all circuits of a directed graph using johnson’s algorithm and OCaml implementation by pietro abate,” 2013. [Online]. Available: https://github.com/josch/cycles_johnson_abate
[36] M. Pezzè and M. Young, Software Testing and Analysis: Process, Principles, and Techniques. Hoboken, NJ, USA: Wiley, 2008.
[37] 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.
[38] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, “Henshin: Advanced concepts and tools for in-place EMF model transformations,” in Proc. Int. Conf. Model Driven Eng. Lang. Syst., 2010, pp. 121–135.
[39] R. Shahin and M. Chechik, “Automatic and efficient variability-aware lifting of functional programs,” Proc. ACM Program. Lang., vol. 4, no. OOPSLA, Nov. 2020, Art. no. 137.
[40] T. Reps, S. Horwitz, and M. Sagiv, “Precise interprocedural dataflow analysis via graph reachability,” in Proc. 22nd ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang., 1995, pp. 49–61.
[41] 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.
[42] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, “Using datalog with binary decision diagrams for program analysis,” in Proc. Asian Symp. Program. Lang. Syst., 2005, pp. 97–118.
[43] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, Horn Clause Solvers for Program Verification, Berlin, Germany: Springer International Publishing, 2015, pp. 24–51.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.