Applying Declarative Analysis to Industrial Automotive Software Product Line Models

Ramy Shahin (✉ r.shahin@computer.org)

University of Toronto - St George Campus: University of Toronto <https://orcid.org/0000-0001-8724-3934>

Rafael Toledo

University of Waterloo

Robert Hackman

University of Waterloo

Ramesh S

: General Motors Corp

Joanne M. Atlee

University of Waterloo

Marsha Chechik

University of Toronto

Research Article

Keywords: Software Product Lines, Automotive, Declarative Analysis, Visualization

Posted Date: January 27th, 2022

DOI: <https://doi.org/10.21203/rs.3.rs-1264268/v1>

License: This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Applying Declarative Analysis to Industrial Automotive Software Product Line Models

Ramy Shahin · Rafael Toledo · Robert Hackman · Ramesh S · Joanne M. Atlee · Marsha Chechik

the date of receipt and acceptance should be inserted later

Conflict of Interest. The authors have no relevant financial or non-financial interests to disclose.

Abstract Software Product Lines (SPLs) are families of related software products developed from a common set of artifacts. Most existing analysis tools cannot be applied to an entire SPL, but rather must be applied an SPL's products one at a time. Some tools have been redesigned or re-implemented to support the kind of variability exhibited in SPLs, but this usually takes a lot of effort and is error-prone. Declarative analyses written in languages like Datalog have been collectively *lifted* to SPLs in prior work, which enables the application of existing declarative analyses to SPLs.

In this paper, we apply five declarative analyses (*behaviour alteration*, *re-cursion analysis*, *simplifiable global variable analysis*, and two of their variants)

Ramy Shahin
University of Toronto
E-mail: rshahin@cs.toronto.edu

Rafael Toledo
University of Waterloo
E-mail: rftoledo@uwaterloo.ca

Robert Hackman
University of Waterloo
E-mail: r2hackma@uwaterloo.ca

Ramesh S
General Motors
E-mail: ramesh.s@gm.com

Joanne M. Atlee
University of Waterloo
E-mail: jmatlee@uwaterloo.ca

Marsha Chechik
University of Toronto
E-mail: chechik@cs.toronto.edu

to a set of automotive software product lines from General Motors. We discuss the design of the analysis pipeline used in this process, present its scalability results, and provide a means to visualize the analysis results for a subset of products filtered by feature expression. We also reflect on some of the lessons learned throughout this project.

Keywords Software Product Lines · Automotive · Declarative Analysis · Visualization

1 Introduction

Software Product Lines (SPLs) are families of related products, usually developed together from a common set of artifacts. Each product (also called a configuration) is a combination of features, each of which is either present or absent in the product. Due to this combinatorial nature of features, the number of potential products grows exponentially with the size of the SPL’s feature set. The high level of configurability within an SPL is usually desired as a means to support mass customization of products. However, analysis tools (such as syntax analyzers, type checkers, model checkers, static analysis tools) are typically designed to work on a single product, not a whole SPL. Applying an analysis to each product separately is usually infeasible for non-trivial SPLs because of the exponential number of products (Liebig et al., 2013).

Since all products of an SPL share a common set of artifacts, analyzing each product individually (usually referred to as *product-based analysis* (Thüm et al., 2014)) would involve a lot of redundant computations. Finding ways to leverage the high degree of commonality across products 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 (Bodden et al., 2013; Classen et al., 2010; Gazzillo and Grimm, 2012; Kästner et al., 2011, 2012; Midtgaard et al., 2015; Salay et al., 2014). The resulting *variability-aware analyses* (Thüm et al., 2014) analyze an SPL as a whole and show significant time savings compared to the product-based analyses on the SPL’s set of products. The downside is the amount of effort required to correctly lift each of those analyses.

In previous work, we lifted a whole class of analyses as opposed to a single analysis. Specifically, we designed and implemented a *variability-aware* Datalog engine (Shahin and Chechik, 2020b) that can be used to efficiently apply an existing single-product Datalog analysis to a fact-based model of a whole product line (Shahin et al., 2019); and we used the engine to apply lifted pointer and taint analyses to Java product lines (Shahin et al., 2019) and other lifted analyses to C-language product lines (Shahin et al., 2021a). Our approach applies to *annotative* SPLs, in which each element in an SPL artifact is annotated with the features to which it belongs. In this project, we leverage the variability-aware Datalog engine to lift a set of five program analyses, and we apply each analysis to seven automotive controller product lines

provided by General Motors. This project also considers how to visualize the analysis results, so that the engineer can easily explore the results from the perspective of different sets of products. Other efforts to visualize the results of a variability-aware analysis aim to support the optimization of the SPL configuration Loesch and Ploedereder (2007), the inspection of consequences from configuration decisions Botterweck et al. (2008), and comprehension of class diagrams of an SPL Strüber et al. (2020). Our project differs from previous works by focusing on the comprehension of variability-aware program analyses, in particular control- and data-flow analyses.

This paper makes the following contributions: (1) We outline the design of a pipeline for variability-aware analysis of C-language product lines. (2) We present the results of applying a set of program analyses using our pipeline to a set of automotive software product lines from General Motors. (3) We discuss the lessons learned throughout the project.

Early results of this work were published in the Practice and Innovation Track of MODELS 2021 (Shahin et al., 2021b). This paper extends this earlier work in the following ways: (1) We outline five different declarative analyses written in Datalog and present the results of applying them to a set of industrial software product lines from General Motors. Our evaluation compares the performance of analyzing the whole product line against analyzing the configuration that includes all features. Only one of these analyses (behavior alteration) was covered in (Shahin et al., 2021b). (2) We provide a detailed description of the interactive visualization and filtering of results. (3) We present the results of applying the extended set of analyses to seven product lines from General Motors, as opposed to the six product line models described in (Shahin et al., 2021b).

The rest of the paper is organized as follows. Section 2 provides a background on SPLs, Datalog, and lifted declarative analyses. In Section 3, we present our five declarative analyses of interest. In Section 4, we present our interactive visualizer to support the exploration of analysis results. In Sections 5 and 6, we present our industrial examples and the results of applying our lifted analyses to them, respectively. We discuss lessons learned in Section 7, present related work in Section 8, and conclude in Section 9.

2 Background

In this section, we briefly define the concepts we build upon in the rest of the paper. In particular, this includes background on software product lines, declarative analyses of relational models, the behavior alteration analysis, and visualization of variability results.

2.1 Software Product Lines

A *Software Product Line (SPL)* is a family of related software products, developed together from a common set of artifacts (Clements and Northrop,

```

1  extern int GlobVar; //
2  shared from C2
3  extern bool FA; // Feature
4  variable
5  extern bool FB; // Feature
6  variable
7  ...
8  class A {
9      int x = 0;
10
11      int updateX() {
12          if (FA) {
13              x = GlobVar * 2;
14              if (FB) {
15                  x++;
16              } else { // !FB
17                  x = (++
18                      GlobVar) * 2;
19                  // FB
20              }
21              ...
22          } // FA
23      }
24      ...
25  }
26
27  (a) Component C1.

```

```

1  int GlobVar = 0; // shared
2  global
3  extern bool FB; // Feature
4  variable
5
6  int foo() {
7      ...
8      if (FB) {
9          return GlobVar;
10     }
11 }
12
13 int bar() {
14     ...
15     if (GlobVar > 20) {
16         return foo();
17     }
18 }
19
20 (b) Component C2.

```

Fig. 1: An example of a Software Product Line with features FA and FB, and components C1 and C2.

2001). The unit of variability in an SPL is a *feature*, where each feature can be either present or absent in each of the individual products. Because of the combinatorial nature of SPL features, the number of products grows exponentially with the number of features. However, there are typically constraints among features that preclude all possible feature combinations from generating valid products. A Feature Model (Kang et al., 1990) captures the set of valid feature combinations.

For example, the SPL in Figure 1 has two features, FA and FB. FA and FB are assumed to be compile-time Boolean constants, each indicating whether its corresponding feature is present or absent in the product. Feature-specific code is guarded within conditional statements, with *feature expressions* (propositional formulas over features) as conditions. This example is usually classified as an *annotative* SPL (Thüm et al., 2014) because different program artifacts (lines of source code in this case) can be individually annotated with feature expressions. The whole program (the union of all features) is usually referred to as the *100% representation* of the product line (because it is generally true that there is no valid product that includes all features, due to feature constraints).

```

1  transVarWrite(v0, v1) :- varWrite(v0, v1).
2  transVarWrite(v0, v2) :- varWrite(v0, v1),
3                             transVarWrite(v1, v2).
4
5  behAlter(f0, f1) :- write(f0, v0),
6                      transVarWrite(v0, v1),
7                      varInfFunc(v1, f1),
8                      cFunction(f0, c0),
9                      cFunction(f1, c1),
10                     c0 != c1.

```

Fig. 2: Datalog program for detecting symptoms of behaviour alterations.

This annotation mechanism allows assigning a given code block a *Presence Condition (PC)*, which specifies a feature expression denoting the set of products in which the line exists. For example, in Figure 1a, the PC of line 10 is **FA**, meaning that line 10 exists in all, and only in, products that include feature **FA**. Line 13, on the other hand, has $(\mathbf{FA} \wedge \mathbf{FB})$ as a PC because both features **FA** and **FB** need to be included in a product for this line to exist. Similarly, the PC of line 15 is $(\mathbf{FA} \wedge \neg \mathbf{FB})$, indicating that this line only exists in products including feature **FA**, and excluding feature **FB**.

A single software product can be generated from an SPL given a *feature configuration*, which specifies the set of features to be included in the product to be generated. For example, in Figure 1a, the feature configuration $\{\mathbf{FA}\}$ would generate a program with all code blocks except for line 13 because **FB** is not included in the configuration. The configuration $\{\mathbf{FA}, \mathbf{FB}\}$, on the other hand, would generate all blocks of code except for line 15, which would only exist in products where **FA** is present and **FB** is absent. We refer to the product generated from product line $\mathcal{L}$ and feature configuration ρ as $\mathcal{L}|_{\rho}$.

The primary motivation behind developing a family of products together as an SPL instead of developing each product independently is to maximize reuse of common software artifacts across products, leveraging the potentially high degree of commonality among them. Different techniques of developing SPLs have been proposed and used in practice (Gacek and Anastasopoulos, 2001; Apel and Kaestner, 2009; Schaefer et al., 2010).

A typical software development life cycle also includes the use of various tools to perform a variety of analyses of software artifacts. These include tools for bug-finding, metric generation, and performance assessment. In most cases, the tools can be applied only to one software product at a time rather than to the entire SPL. The naive approach of generating each and every product and applying an analysis tool to it individually is usually infeasible because of the exponential growth in the number of products as the number of features increases.

2.2 Datalog-Based Analysis

We express analyses in Datalog. Datalog (Ceri et al., 1989) is a logic programming language that supports rule-based inference over relational data. Program analyses written in Datalog are applied to relational facts extracted from programs. A Datalog program is a set of rules, with a set of premises (the *body* of the rule), and a conclusion (the *head* of the rule). Figure 2 shows a Datalog program (simplified for presentation purposes) for detecting symptoms of behaviour alteration between software components. For example, line 1 of Figure 2 is a rule with a single clause in the body (the **varWrite** clause), and the **transVarWrite** is the head (conclusion) of the rule.

Lines 1-3 compute the transitive closure of the **varWrite** relationship, thereby finding all data-flows in which one variable is used in the assignment expression of another variable (including parameter assignments). Lines 5-10 define behaviour alteration as a data-flow (**transVarWrite**) that starts with a variable assignment (**write**) in function **f0**, and ends with function **f1** whose invocation is influenced by a variable value (**varInfFunc**). As we are only interested in behavior alterations crossing component boundaries, we exclude intra-component alterations (lines 8-10).

Running the analysis on the facts extracted from the code in Figure 1 yields that the function **updateX** in component **C1** may influence whether the function **foo** in **C2** is called: (i) **write** relationship from function **updateX** to variable **GlobVar** (line 15 in **C1**); (ii) **varWrite** relationship from the variable **GlobVar** to itself (line 15 in **C1**); (iii) **varInfFunc** relationship from the variable **GlobVar** to the function **foo** (lines 12-13 in **C2**).

2.3 Lifted Declarative Analyses

Several software analyses have been re-designed and implemented to support efficiently analyzing the whole SPL at once (more on this in Section 8). Those are usually referred to as *variability-aware analyses*, and the process of transforming a single-product analysis to an variability-aware analysis is usually referred to as *variability-aware lifting* (Bodden et al., 2013; Salay et al., 2014; Shahin et al., 2019; Shahin and Chechik, 2020a). A *lifted* analysis is expected to preserve the semantics of its single-product counterpart, while tracing each of the results of the analysis to the set of products to which it applies. We use the notation $f^\uparrow$ to refer to a lifted version of a product-based analysis f .

For example, assume that we are designing an analysis that detects interference between software components through the use of global variables. Component **X** interferes with component **Y** if **X** writes to some global that is read by **Y**. The example SPL in Figure 1 shows two components, **C1** and **C2**, both of which access a global variable **GlobVar**. Component **C2** (Figure 1b) reads the value of **GlobVar** in lines 7 and 12, but the first read exists only in products that include feature **FB**, while the second exists in all products. Component **C1** (Figure 1a) reads from **GlobVar** in lines 10 and 15, and it also

Fig. 3: End-to-end fact extraction and analysis pipeline.

writes to `GlobVar` in line 15. The presence condition of line 15 is $(FA \wedge \neg FB)$, which means that `C1` interferes with the first read of `C2` (line 7); and if `FB` is included in the product, it might also affect the control-flow of function `bar` depending on the value of `GlobVar` at line 12.

A variability-aware analysis is expected to report the results of applying the corresponding product-based analysis to each product configuration, annotating each of the results with its correct product configuration. For the example in Figure 1, it is not enough to report that `C1` interferes with `C2`. Instead, the analysis needs to specify the product configuration(s) in which this interference occurs ($\{FA\}$ in this case).

Formally, given a product line $\mathcal{L}$ and a lifted analysis function $f^\uparrow$, applying $f^\uparrow$ to $\mathcal{L}$ and restricting the results to a single product denoted by product configuration ρ should be the same as applying the original analysis f to the set of artifacts belonging only to the product configuration ρ (Shahin and Chechik, 2020a).

Instead of re-implementing a given analysis to make it variability-aware, another approach is to lift the language in which the analysis has been implemented. This has the advantage of not having to modify the original product-based analysis at all, and if many analyses are written in the lifted language, they all get lifted for free. For example, analyses written in Datalog have been collectively lifted (Shahin et al., 2019) by extending the Datalog language with optional presence condition annotations at the fact level, and implementing a variability-aware fact inference algorithm in the `Soufflé↑` (lifted `Soufflé`) Datalog engine (Shahin and Chechik, 2020b).

3 The Analysis Pipeline

We implemented an end-to-end pipeline for extracting a product line model from source code, analyzing it, and interactively visualizing the results. The analysis pipeline integrates components used in previous projects (Shahin et al., 2019; Muscedere et al., 2019), together with some adapter components for converting data from one format to another. The overall pipeline design is shown in Figure 3.

Fig. 4: The fact extraction of component C1.

An SPL model is extracted from C-language source files using a new variability-aware version of *Rex*, which extracts syntactic facts about the source files (e.g., variable declarations, variable assignments, function declarations, function calls) and annotates a fact with a presence condition (PC) if the fact relates to code that is present in a subset of products. Facts and their presence conditions are extracted in *Tuple-Attribute (TA)* format (Holt, 1997), which are then converted to Datalog fact format using the *ta2tsv* adapter component.

The analyses of interest are expressed as a collection of Datalog rules. Datalog facts and rules become inputs to Soufflé[†], a variability-aware Datalog engine. The output of Soufflé[†] can be optionally filtered using a Feature Model, removing those facts that do not belong to any of the valid products of the SPL. We describe the analysis-related components of this pipeline below. The visualization component is explained in Section 4.

3.1 Variability-Aware Fact Extraction

In order to support analysis of SPL models, we developed a variability-aware version of *Rex* that annotates entities and relationships with their presence conditions. A *Rex* user can specify, by type and naming convention, which program variables are to be considered feature variables to be used in presence conditions (e.g., only constant global `bool` or `enum` type variables). Variability-aware *Rex* keeps track of all relevant predicates currently in effect while walking the AST and uses that information to annotate the model as it is extracted.

Figure 4 gives an overview of the *Rex* extraction process of the component C1 in Figure 1a. On the left is the input C++ code, the middle of the figure depicts the extracted information as an in-memory hierarchical graph, and on the right is the generated TA model. In this example, *Rex* creates model nodes for the class `A`, function `updateX`, and variables `x`, `FA`, `FB`, and `GlobVar`¹. Each

¹ The names of the entities are simplified for this example to improve legibility. In practice, *Rex* creates long identifier names that capture the entities context (i.e., enclosing function, class, etc., up to and including filename).

`contain` edge corresponds to an entity declaration (e.g., class `A` contains the declaration of variable `x`). When one variable appears in an expression that is assigned to another variable (e.g., the use of `GlobVar` in an assignment to variable `x` in `C1`), a `varWrite` edge is created from the used variable to the assigned variable (e.g., `varWrite GlobVar x`). The creation of the other edges follows the same pattern. Attributes of entities and relationships are listed at the end of the TA model. The attribute `PC` records presence conditions: any entity or relationship that is annotated with a `PC` attribute represents a fact that is conditionally present in the model, depending on the value of the `PC`'s feature variables. Thus, variability-aware Rex extracts a 150% model representing facts from all products in the SPL, where conditional facts are annotated with their products' presence conditions.

In the last step of the extraction, the in-memory hierarchical graph is converted to a TA model, which is a textual representation of the graph. Because identifiers must be declared before being used in the model, the output lists: (i) all graph nodes (representing entities), (ii) all graph edges (representing relations), and finally (iii) all attributes of nodes and edges. The tuple representations of nodes, edges, and attributes have the following structure:

```
$INSTANCE <NODE_ID> <NODE_TYPE>
<EDGE_TYPE> <EDGE_SOURCE> <EDGE_TARGET>
<ID> { <KEY> = <VALUE> . . . }
```

Because of the nature of static analysis, the resulting model is an over-approximation of the program's actual set of facts: it may contain some facts that are infeasible (e.g., a function call in a conditional branch that never executes).

Facts are ported automatically to the TSV format using the *ta2tsv* command-line tool that we wrote specifically for that purpose. In a TA model, presence conditions are not co-located with their associated facts, but rather are listed as attributes at the end of the file. Our *ta2tsv* command-line tool associates the presence-condition attributes with their corresponding TSV records.

3.2 Analyses of Interest

In collaboration with General Motors, we identified three analyses of interest, *behaviour alteration*, *recursion*, and *simplifiable-global variable*, and applied them to the industrial case study. Each of these analyses was originally applicable to a model of a single product, not a product line. We aimed to produce lifted analyses that operate on a 150% factbase and return the same set of results as that computed by the original analysis when applied to each product configuration. The lifted analyses are also expected to annotate each of the results with a presence condition indicating the set of products to which this particular result applies.

Instead of adapting the analyses to become variability-aware, we decided to use the existing variability-aware Datalog engine Soufflé[†] (Shahin and Chechik, 2020b). This way we were able to leverage all the optimizations in Soufflé[†] to

1 ensure the scalability of our solution to industrial-scale systems, and at the
2 same time minimize the effort needed to build the components of the analysis
3 pipeline. We were also able to leverage the flexibility of using Datalog as a
4 query language for expressing analyses.

5 Soufflé[†] takes as input facts annotated with precedence conditions and
6 infers additional facts based on a set of Datalog rules (the analysis logic in
7 our case). Presence conditions of inferred facts are calculated as a part of
8 the inference process. Those presence conditions are also checked for propo-
9 sitional satisfiability. Because an *unsatisfiable* PC indicates an empty set of
10 products, inferred facts with unsatisfiable PCs are removed from the factbase.
11 Soufflé[†] stores presence conditions as Binary Decision Diagrams (BDDs). This
12 has the advantage of keeping a canonical representation of each presence condi-
13 tion, eliminating redundancies due to propositional logic identities (e.g., com-
14 mutativity of conjunction and disjunction). CUDD (Somenzi, 1998), the BDD
15 package used by Soufflé[†], also caches BDDs, saving time on BDD construction.
16
17

18 3.2.1 Behaviour Alteration Analysis

19
20 A *behaviour alteration* (Muscedere et al., 2019) is a form of data-flow *compo-*
21 *nent interaction* that occurs when a change to a variable value made in one
22 component alters the behaviour of another component. Such an analysis is
23 useful in large component-based systems, where an engineering team knows
24 its components well, but does not know all of the ways in which actions taken
25 in its components can affect the behaviours of other teams' components. The
26 specific instance of behaviour alteration used in this paper is (1) an assignment
27 made in component C1 to a variable *v*, (2) whose value impacts other vari-
28 ables through variable assignments, and impacts other components through
29 parameter passing; until (3) a variable *x* whose value has been influenced by
30 the modified value of *v* is used in the decision condition of some control struc-
31 ture (i.e., an *if*, *for*, *while*, or *switch* statement) in another component Cn
32 that (4) guards a function call. Thus, the analysis looks for a data flow from a
33 variable assignment in one component to a control structure in another compo-
34 nent, where the control-structure's statement block includes a function call.
35 Figure 1 gives a simple example where the write to *GlobVar* in line 15 of compo-
36 nent C1 could affect whether or not the function *bar* calls the function *foo*
37 in component C2.
38
39

40 The analysis operates on extracted "facts" about C/C++ source code,
41 rather than operating on the code itself, to enable the analysis to scale to
42 large software systems. Specifically, a fact extractor Rex (Muscedere et al.,
43 2019), based on the *Clang++* open-source compiler², parses C/C++ source-
44 code files, generates abstract syntax trees (ASTs), and extracts facts of interest
45 from the AST into an in-memory hierarchical graph. Source-code entities such
46 as variable declarations and function declarations are the nodes of the graph;
47 and relations such as variable assignments (in which one variable is used in
48

49 ² <https://clang.llvm.org/index.html>

the assignment expression for another variable), function calls, and containment (of variable declarations within functions, function declarations within files, components comprising files) are the edges of the graph. Additional information about the nodes and edges are recorded as associated attributes. Rex outputs the resulting graph as a collection of facts (called a *factbase*) about source-code entities, their relations, and their respective attributes represented as three-tuples (triples) in the Tuple-Attribute (TA) language (Holt, 1997).

To analyze the code provided by General Motors, we developed a specialized version of the behavior alteration analysis, henceforth called *the GM variant*. This analysis has an additional requirement that a particular middleware component, which handles inter-component communications, cannot be the start- or the end-point of a behaviour alteration path. This component is expected to communicate with multiple other components, and thus behaviour alteration paths that start or end with this component are uninteresting and would clutter the analysis results.

3.2.2 Recursion Analysis

We also ran multiple analyses to detect occurrences of recursion in the provided source code:

1. direct recursion;
2. indirect recursion;
3. indirect recursion across component boundaries.

The first analysis detects functions that directly call themselves; the second analysis detects functions that indirectly call themselves (via a cycle of function calls); and the third analysis detects functions that indirectly call themselves, where at least two functions in the call cycle reside in distinct components.

These analyses exemplify simple coding standards, like MISRA C³, that are used in the automotive and other safety-critical industries, that can be expressed easily as Datalog queries. Code that is compliant with the MISRA C standard not expected to have any recursion.

3.2.3 Simplifiable Global Variable Analysis

A *simplifiable global variable* is a global variable that is used only to pass data to a single function. If a global variable is *simplifiable* then it can likely be replaced with a parameter of the function that reads from it. In general, global variables can introduce needless coupling of components and potential logical errors in maintaining their state; thus General Motors was interested in detecting global variables that could be refactored as parameters.

Specifically, a global variable is *simplifiable* if it meets two conditions:

1. The global variable v is read by only one function f .
2. All functions that call function f write to global variable v before calling f .

³ <https://www.misra.org.uk/>

If these conditions are met, the global variable v can potentially be refactored as a parameter to the function f , given that the variable does not maintain state and instead is always overwritten by a calling function before that function calls f . Figure 5 illustrates a simplifiable global variable `CtrlIdx`, where functions `X` and `Z` are the only functions in the program that call function `Y`.

```

1  int CtrlIdx;
2
3  void Y() {}
4      ...
5      if (CtrlIdx==0) ...
6      if (CtrlIdx==1) ...
7      ...
8  }
9  void X() {}
10     CtrlIdx = 0;
11     ...
12     Y();
13     ...
14 }
15 void Z() {
16     CtrlIdx = 1;
17     ...
18     Y();
19     ...
20 }

```

Fig. 5: An example of a simplifiable global variable `CtrlIdx` that may be simplified to be a parameter of function `Y`.

4 Interactive Visualization

Our pipeline includes an interactive visualizer that supports inspection of the analysis results by visually encoding which facts and analysis results belong to which software products. Because the results of a lifted analysis are inferred *paths* in the factbase, they can be portrayed as edges in a graphical model representing the analysis results. Although a graphical model can concisely represent the analysis results, the task of understanding how the results apply to specific SPL configurations requires the engineer to read and compare presence conditions on multiple edges. The interactive visualizer enables the engineer to apply colored filters to the results to help identify groups of paths occurring in related software products.

Our visualizer is implemented on top of the Neo4j Browser⁴, the user interface provided by the open-source graph database Neo4j⁵. As a database

⁴ <https://neo4j.com/developer/neo4j-browser/>

⁵ <https://neo4j.com/>

Fig. 6: Neo4j Browser interface

engine, Neo4j enables the storing and querying of graphical data like the facts and results of our analyses, so we import our results into an instance of the Neo4j database so that it can be queried and visualized. The Neo4j Browser presents the analysis results in visualization frames (see Figure 6). Each visualization frame has an central interactive area where the user can explore the displayed information and rearrange the layout of the graph.

The visualization frames include an expandable sidebar that gives an overview of the data being represented and enables the customization of the graph visualization. Some of the visual parameters that can be changed by the user are the nodes size, the edges thickness, and the information presented on the labels. For example, in Figure 6, edges between $\langle f2, v2 \rangle$, $\langle v2, v1 \rangle$, and $\langle v1, f1 \rangle$ display the presence condition for each interaction, whereas the edge between $\langle f2, f1 \rangle$ shows the type of the path inferred by the lifted analysis. Each presence condition labelling an edge indicates the software products for which that relationship applies.

We have enhanced the Neo4j Browser to support the exploration of our analysis results based on user-specified filters (see Figure 7). The engineer specifies a filter as a presence condition representing a set of SPL configurations (see Figure 7-A), and the visualizer highlights the subset of results that satisfy the filter's presence condition. Our visualizer employs Logic Solver⁶, a boolean satisfiability solver, to reason for each fact whether the fact's presence condition satisfies also a filter's presence condition. The visualizer automatically assigns a distinct color to the edges that satisfy the filter, thereby preserving the original results as well as highlighting the filtered results.

Multiple filters can be applied to the same analysis results, resulting in a colour-coded graph visualization that highlights which analysis results pertain to specific configurations. After applying filters, the engineer can inspect the types of edges and color mapping for the created filters in the bottom-left

⁶ <https://github.com/meteor/logic-solver>

Fig. 7: Three main sections of the behavior alteration analysis result visualizer: (A) the text box that allows the user to create filters with feature expressions, (B) the legend box showing the colour and shape of the links mapped to each filter, (C) the overview sidebar where the user can customize the visual parameters like colour, size, and shape of the links and nodes.

corner of the screen (see Figure 7-B) and customize the visual parameters for each filter in the sidebar (see Figure 7-C). For each edge in the graph, the visualizer displays both of the type of relationship the edge represents and the edge's presence condition. The relationship type appears in bold whereas the presence condition is located on the opposite side of the edge. For example, as shown in Figure 7, the edge between nodes **f1** and **f2** reports an alteration behaviour that originates in function **f2** and manifests in function **f1**; this alteration behaviour is present only in products that satisfy the presence condition $FA \wedge FB \wedge FC \wedge FD$.

Through the application of one or more filters, the engineer can explore and better understand the analysis results. A single filter can be used to determine and visualize which results apply to a particular set of software products of interest. Alternatively, the engineer can compare how two or more sets of software products differ with respect to the analysis results by applying multiple

Fig. 8: The visualization of a subset of the analysis results and the effects of (a) applying an initial filter (yellow edges) showing the analysis results that apply to a given program configuration and (b) applying a second filter (blue edges) that adds a new feature to the original configuration.

filters, one for each product set. Moreover, they can see the effects of adding or removing a single feature from a product set by applying filters that include or exclude the feature of interest and seeing which results are highlighted by the different filters.

Figure 8(a) shows an example of creating a single filter to identify the analysis results that apply to a specific set of software products. Figure 8(b) shows the effects of applying a second filter to the same visualization to assess the impact of adding a feature to the first filter. The edges colored yellow highlight the analysis results that satisfy only the first filter and the edges colored blue and yellow highlight the analysis results that hold for the larger set of products (satisfying both the first and second filters)⁷.

5 Industrial SPL Examples

Our industrial study was performed on models extracted from seven vehicle controller product lines provided by General Motors, which are abstractly named SPL-A, SPL-B,..., SPL-G to obfuscate sensitive industrial data. Metrics on the sizes of all seven product lines are shown in Table 1. For example, SPL-A has 5431 header (.h) files, with a total of 350,102 lines of code (LOC). It also has 5133 C language source files (.c), totalling 730,947 lines of code.

The General Motors controllers encode the inclusion or exclusion of features using *configuration parameters*. Configuration parameters are represented as global constants of enumerated types (`enum`) or boolean type (`bool`); as such, the configuration parameters represent SPL variability in a way that enables

⁷ Node labels and presence conditions on edges have been omitted from Figure 8(b) to avoid revealing proprietary information.

Table 1: Size metrics for the seven product lines analyzed. For each product line, we list the number of header files (.h) and C source files (.c), together with the total number of Lines of Code (LOC).

	SPL-A	SPL-B	SPL-C	SPL-D	SPL-E	SPL-F	SPL-G
(.h) Files	5431	6277	4702	6292	5243	6115	8137
(.h) LOC	350,102	570,174	285,132	586,985	337,946	572,851	759,160
(.c) Files	5133	6826	4300	6943	4981	6464	8458
(.c) LOC	730,947	1,016,063	750,000	979,466	752,669	1,088,811	1,639,822

easy configurability as to which features are present in any particular vehicle. The values of these parameters are defined at deployment time during vehicle manufacturing (Young et al., 2017).

Such an encoding of variability means that the source code includes all of the code relevant to all features. Thus, each controller code-base is a 150% representation of the controller’s SPL, and an individual controller product is configured by setting the values of these configuration parameters. The SPL in Figure 1 is a demonstration of this kind of variability representation, where the `bool` variables `FA` and `FB` are examples of configuration parameters.

For some of our analyses (specifically, behaviour alteration analyses and one of the indirect recursion analyses), the most interesting results are the paths between functions that reside in different components. However, there is no identifiable notion of a *component* unit in C/C++ source code (components in the General Motors controllers are made up of collections of source-code files, so we cannot use compilation units as the delimiters of components). Instead, General Motors shared with us a high-level decomposition of their code into components, and we incorporated this information into a controller’s factbase as additional facts: we introduced a *component* entity fact for each distinct component and a *contains* relationship fact between each component entity and its constituent source files. The additional facts allowed us to adapt our analyses to avoid reporting intra-component results (see the last line in Figure 2).

6 Applying Analysis to the Industrial Examples

One of the primary goals of this project was to validate that the variability-aware Datalog analysis approach (Shahin et al., 2019) is scalable to real-life industrial SPLs. We informally define scalability as having a marginal performance overhead compared to analyzing the 150% representation of the SPL, which implicitly means having an exponential speedup compared to product-based analysis of each single product individually.

For each controller SPL, we used variability-aware Rex to automatically extract a *150% representation* (i.e., a model representing a single product with all features present) and an SPL model, with feature variability represented as presence-condition annotations on facts. We translated the extracted facts into

Table 2: Results of applying the behavior alteration analysis.

	SPL-A	SPL-B	SPL-C	SPL-D	SPL-E	SPL-F	SPL-G
Features	~400	~500	~900	~600	~600	~500	~1600
Facts	157303	225538	215120	228185	227241	226640	621714
VFacts	698	1070	2126	1148	2078	955	3944
VFacts (%)	0.44	0.47	0.99	0.50	0.91	0.42	0.63
150% Time(sec.)	3.93	5.14	17.62	4.81	17.76	4.87	175.17
150% Results	128780	177806	399085	167243	438319	177689	1594370
Distinct PCs	198	278	779	318	1123	278	2225
Time(sec.)	6.49	8.29	28.68	7.79	32.63	7.62	427.60
Results	128759	177801	399027	167241	436740	177684	1592336
VResults	505	676	3484	756	19046	924	127279
VResults (%)	0.39%	0.38%	0.87%	0.45%	4.36%	0.52%	7.99%
Overhead	64.82%	61.41%	62.71%	61.95%	83.68%	56.46%	144.11%

Table 3: Results of applying the GM variant behavior alteration analysis.

	SPL-A	SPL-B	SPL-C	SPL-D	SPL-E	SPL-F	SPL-G
Features	~400	~500	~900	~600	~600	~500	~1600
Facts	173454	241689	231271	244336	243392	242791	637865
VFacts	698	1070	2126	1148	2078	955	3944
VFacts (%)	0.40	0.44	0.92	0.47	0.85	0.39	0.62
150% Time(sec.)	3.99	5.17	17.72	4.83	17.84	4.88	173.06
150% Results	142457	191403	411397	180496	454691	191423	1607902
Distinct PCs	197	276	771	317	1111	276	2213
Time(sec.)	6.57	8.30	28.60	7.85	32.05	7.60	410.06
Results	142436	191398	411339	180494	453095	191418	1605848
VResults	527	690	3253	768	18796	938	127522
VResults (%)	0.37%	0.36%	0.79%	0.43%	4.15%	0.49%	7.94%
Overhead	64.77%	60.72%	61.34%	62.43%	79.69%	55.59%	136.94%

Table 4: Results of applying the simplifiable global variable analysis.

	SPL-A	SPL-B	SPL-C	SPL-D	SPL-E	SPL-F	SPL-G
Features	~400	~500	~900	~600	~600	~500	~1800
Facts	228473	333016	285853	334635	303877	338233	769236
VFacts	710	1009	1624	1080	1693	879	3160
VFacts (%)	0.31	0.30	0.57	0.32	0.56	0.26	0.41
150% Time(sec.)	4780.88	9377.31	4672.73	9526.3	5859.57	9615.21	25967
150% Results	129629	178061	145877	177084	151796	177988	412600
Distinct PCs	245	355	755	392	557	341	2328
Time(sec.)	5102.92	10032.1	5090.34	10225.2	6280.87	10381.3	28618
Results	129649	178079	145904	177110	151840	178005	412688
VResults	1226	1556	1580	1598	1833	1493	3790
VResults (%)	0.95%	0.87%	1.08%	0.90%	1.21%	0.84%	0.92%
Overhead	6.74%	6.98%	8.94%	7.34%	7.19%	7.97%	10.21%

Datalog facts. For each of the analyses, we applied Soufflé (version 1.3.1) to the 150% representation, and Soufflé[†] to the factbase annotated with presence conditions. We repeated each analysis experiment five times, reporting the average execution time after excluding the minimum and maximum times.

Tables 2-6 summarize the results of our experiments, one for each analysis. For a given SPL, each analysis might depend on a different set of input facts (**Facts**). Some of these facts are variational (**VFacts**), and the percentage of variational facts to the total number of facts is **VFacts (%)**. The total number of features referenced by the presence conditions of the variational facts is **Features** (approximated to the nearest hundreds). For example, when apply-

Table 5: Results of applying the recursion analysis.

	SPL-A	SPL-B	SPL-C	SPL-D	SPL-E	SPL-F	SPL-G
Features	~300	~300	~600	~300	~400	~300	~1200
Facts	51264	72154	65373	74314	80767	73521	163629
VFacts	292	420	756	426	569	356	1097
VFacts (%)	0.57	0.58	1.16	0.57	0.70	0.48	0.67
150% Time(sec.)	1.46	1.93	2.00	2.01	2.72	1.84	6.56
150% Results	285229	347712	358868	364255	502085	332209	1244323
Distinct PCs	199	264	1158	264	360	247	793
Time(sec.)	2.17	2.74	3.39	2.94	3.78	2.57	23.26
Results	285229	347712	358868	364255	502085	332209	1244323
VResults	3533	6960	5503	7223	6224	4328	14119
VResults (%)	1.24%	2.00%	1.53%	1.98%	1.24%	1.30%	1.13%
Overhead	48.44%	42.22%	69.44%	46.10%	38.93%	40.18%	254.36%

Table 6: Results of the cross-module recursion analysis.

	SPL-A	SPL-B	SPL-C	SPL-D	SPL-E	SPL-F	SPL-G
Features	~300	~300	~600	~300	~400	~300	~1200
Facts	111509	158174	117422	162635	142584	162281	309441
VFacts	292	420	954	426	715	356	1097
VFacts (%)	0.26	0.27	0.81	0.26	0.50	0.22	0.35
150% Time(sec.)	1.60	2.09	2.19	2.17	3.40	2.02	7.86
150% Results	315726	382327	424390	398987	681000	369020	1565191
Distinct PCs	140	191	776	192	354	175	643
Time(sec.)	2.29	2.95	3.60	3.01	4.35	2.73	22.78
Results	315726	382327	424390	398987	680999	369020	1565191
VResults	3694	7811	11033	8106	17204	4556	17234
VResults (%)	1.17%	2.04%	2.60%	2.03%	2.53%	1.23%	1.10%
Overhead	42.57%	40.99%	64.11%	38.42%	27.89%	34.81%	189.91%

ing the behavior alteration analysis (Table 2) to SPL-A, the number of input facts relevant to behavior alteration is 157303, 698 of which are variational, corresponding to 0.44%.

The software of a vehicle has many variation points and thus configuration involves many configuration parameters (Young et al., 2017). In our SPL examples, the code has several hundred configuration parameters (**Features** in Table 2 - Table 6) in each of the controllers. Because the number of possible products is exponential in the number of configuration parameters, the large number of configuration parameters makes analyzing individual products infeasible.

When applying the behavior alteration analysis to the 150% representation of SPL-A using Soufflé, analysis time is 3.93 seconds, and the number of output facts is 128780. However, when applying the same analysis to the variational facts of the same product lines using Soufflé[†], analysis time is 6.49 seconds (an overhead of only 64.82%). Soufflé[†] generates 128759 output facts, 505 of which are variational, corresponding to 0.39%. The number of distinct presence conditions calculated as part of the analysis is 198.

The execution times of the different analyses on the 150% representation models range from a few seconds to thousands of seconds in the case of the global passing analysis. Variational analysis time overheads range from 6.74% (global passing applied to SPL-A) to 254.36% (recursion checking ap-

plied to SPL-G). Recall that the cost of product-based analysis, where each product of an SPL is analyzed separately, grows exponentially with the number of features (Liebig et al., 2013). Yet the execution-time overhead of our variability-aware analyses does not seem to correlate with the number of SPL features. The marginal overheads incurred can be considered *very acceptable*, at least in cases like our industry examples, where a system has hundreds of features but sparse variability in terms of the percentage of facts annotated with presence conditions. We also note that for a computation-intensive analysis like global passing, the overheads are significantly lower than those of the other, lighter-weight, analyses. This indicates that the extra costs of presence condition manipulation amortizes over the execution time of the analysis.

In addition to execution time, we also measured the number of facts generated by the analyses, including all intermediate facts generated during inference. There are two reasons behind the discrepancy in the number of facts inferred when analyzing the 150% representation, and when applying Soufflé[†] to variational factbases: (1) variability-aware analysis excludes facts that have unsatisfiable presence conditions, whereas in the analysis of a 150% representation, all inferred facts are deemed to be feasible; and (2) variational aggregator operators (e.g., sum, count) might generate multiple results when applied to a set of facts, while a non-variational aggregator always generates a single result.

We also measured the total number of unique presence conditions computed during the inference process⁸. To our surprise, the number of unique presence conditions in each controller was usually smaller than (and in one case roughly equal to) the number of configuration parameters in the controller – which is far fewer than the number of possible combinations of features. Thus, although the controller’s SPL technically supports an exponential number of configurations (2^N products given N features), the number of variants mentioned in the source code as presence conditions is much smaller. Taking a further look at the presence conditions, we found out that many features always appear together in a presence condition. This kind of feature correlation is not uncommon in SPLs (Apel and Beyer, 2011).

In summary, with a performance overhead of only 6.74%-254.36% compared to the analysis of the single product with all features present (the 150% representation), our evaluation shows that variability-aware analysis scales to large-scale industrial software product lines with hundreds of features.

A secondary product of our work is the use of graph visualization and interactive techniques to enable the engineer to explore and filter the analysis results by feature expression. We hypothesize that colouring the subset of edges associated with a user-provided feature expression increases the readability of the analysis results by highlighting the interactions that match the user’s filter while still offering a view of the facts’ context. This hypothesis and the design of our tool must still be tested by conducting user studies that evaluate the

⁸ This measurement was aided by the fact that the presence conditions have canonical BDD representations.

effect of different visualizations of the filtering and highlighting of analysis results in different scenarios.

Lastly, although we have not had an opportunity to seek detailed feedback from General Motors engineers (e.g., via a qualitative study), we have given multiple presentations to General Motors engineers, including to senior engineers with over 20 years of experience. The engineers continue to express interest in all aspects of the tool chain. In particular, they consider the extracted facts and their variabilities to be extensive, and they see the use of a query language for posing specific and ad-hoc queries over the fact base to be potentially quite powerful. Our presentations of performed analyses often led to questions about followup queries: for example, the recursion results led to questions about identifying the corresponding calling functions; and the lifted analyses over *feature-oriented* software, filtered by presence condition, led to questions about possible queries over facts conditioned on *program variables* subsequently filtered by conditions on variables. Because we express analyses as queries, we were able to perform such followup analyses relatively quickly.

Moreover, the engineers' initial concerns about scalability were quickly alleviated: they were pleasantly surprised to learn that we were analyzing code bases of the sizes of General Motors controllers (i.e., a million lines of code or more). They also felt that the tools' capabilities to report analysis results for *sets* of products (expressed as presence conditions), to filter the results using more-specific presence conditions, and to compare and visualize side-by-side the analysis results for multiple SPL configuration sets were all very promising strategies for reducing the cognitive load on the engineer. More formal experiments evaluating the design of the visualization tool are being planned as future work.

7 Lessons Learned

In this section, we reflect on some of the lessons learned by conducting this project.

7.1 Scalability of Lifted Analysis

In theory, the complexity of software product line analysis is expected to grow with respect to the number of product line features (Liebig et al., 2013). Product variants compose features together, thus the number of product variants typically grows exponentially with the number of features. The idea behind lifting analyses to product lines is to leverage the commonality among different product variants as much as possible to keep the cost of product line analysis reasonable, as opposed to enumerating and analyzing each product variant by itself, which is intractable in most practical cases.

The product lines we analyzed in this study have hundreds of features each, which means that enumerating each product is not an option. The variability-aware overhead reported for Soufflé[†] in earlier work (Shahin et al., 2019) is

marginal, but that was reported for relatively small benchmarks (none of them of industrial scale) of only tens of features each. Results presented in Section 6 show that the performance overhead of full product line analysis using Soufflé[†] is still marginal for industrial product lines, with hundreds of features.

Looking further into the results, the performance overhead does not seem to correlate with the size of the code-base, the size of the extracted model (number of facts), or the number of features of the SPL. This can be explained by differences between the subject SPLs with respect to the code patterns directly relevant to the particular analysis applied. Also measuring the unique number of presence conditions generated throughout the analysis sheds some light on how some features are tightly coupled in industrial product lines, causing the effective complexity of the analysis to be lower than what might be perceived given the number of features.

7.2 Variability Encoding

Soufflé[†] can only handle binary features; that is, a feature can be either present or absent. However, the SPLs we analyzed in this project also encode sets of mutually exclusive features using C-language `enum` data types. For example, if `Feat0`, `Feat1`, `Feat2`, and `Feat3` is a set of four mutually exclusive features, it is a common C-language idiom to encapsulate them in an enumerated data type:

```
1 enum FeatSet {
2     Feat0,
3     Feat1,
4     Feat2,
5     Feat3
6 };
```

Enumerated data types in C are integral types, allowing the use of mathematical integer operators (e.g., addition, conjunction, bit-wise disjunction) and comparison operators on their values. We came across cases where presence conditions included comparison operators on values of enumerated data types, and we had to abstract those predicates into propositional symbols. For example, if `x` is a constant of type `FeatSet`, then the expression `x < Feat2` is a logically valid presence condition, but is not acceptable in Soufflé[†]. We apply a syntactic transformation for these kinds of expressions, turning the above expression into a boolean `x_LT_Feat2`, where the `_LT_` sub-string stands for less-than. We use similar substitutions for other comparison operators.

The fact that the four features belonging to the `FeatSet` are mutually exclusive can be then added to the feature model of the product line. The fragment of the feature model representing this property for `FeatSet` is:

$$\begin{aligned} & \neg (\text{Feat0} \wedge \text{Feat1}) \wedge \neg (\text{Feat0} \wedge \text{Feat2}) \wedge \\ & \neg (\text{Feat0} \wedge \text{Feat3}) \wedge \neg (\text{Feat1} \wedge \text{Feat2}) \wedge \\ & \neg (\text{Feat1} \wedge \text{Feat3}) \wedge \neg (\text{Feat2} \wedge \text{Feat3}) \end{aligned}$$

If a feature from **FeatSet** is mandatory, we also need to add the disjunction of all four features to the feature model:

$$(\text{Feat0} \vee \text{Feat1} \vee \text{Feat2} \vee \text{Feat3})$$

7.3 Variability Annotation

Different techniques have been used to annotate segments of source-code with feature expressions, effectively deciding which pieces of code belong to which features. For example, CIDE (Kästner et al., 2009a) is a colour-based tool that highlights segments of code with different colours, each of which represents a feature. The most commonly used annotation mechanism in industrial product lines is the C Pre-Processor (CPP) (Ernst et al., 2002; Liebig et al., 2010). The CPP provides a high degree of flexibility when annotating source code, allowing for lexical rather than syntactic annotation. This means that any sequence of lexemes (tokens), even if the sequence by itself is not syntactically valid, can be assigned a presence condition. As a result, the 150% representation of an SPL annotated with CPP directives is typically not syntactically well-formed, requiring variability-aware parsing (Gazzillo and Grimm, 2012; Kästner et al., 2011).

The product lines from General Motors, however, use a different annotation mechanism. C-language constants (following a naming convention) are used within the source code to indicate features. Those constants are assigned values as a part of the product configuration process. Feature-specific code is thus enclosed within C-language conditional statements, relying on the compiler to evaluate the compile-time constants at compile time and to eliminate dead-code corresponding to features not included in the product being built.

This annotation technique has two direct consequences. First, while it is less flexible than CPP directives, it does not require variability-aware parsing because the entire product line is a syntactically well-formed C-program. Secondly, existing analysis tools can be applied to the entire product line, in the same way as regular parsers can be applied to it. The downside is that each result of a given analysis is not labeled with the distinct set of products to which it applies. This draws a clear distinction between analyzing the 150% representation of a product line, in the case where it is well-formed and readable by an analysis tool, and variability-aware analysis, where both inputs and outputs of the analysis need to be appropriately annotated.

An indirect consequence of the annotation technique used by General Motors is the possibility of filtering analysis results through user-provided feature expression of interest and presenting only facts with satisfying presence conditions. This capability has the potential to improve the readability of the data and support the experience of the user visually inspecting the analysis results.

8 Related Work

Variability-Aware Analysis. Different kinds of source-code analyses have been re-implemented to be variability aware (Thüm et al., 2014). For example, the TypeChef project (Kästner et al., 2011, 2012) implements variability-aware parsing (Kästner et al., 2011) and type checkers (Kästner et al., 2012) for Java and C. The SuperC project (Gazzillo and Grimm, 2012) is another C language variability-aware parser. With respect to model-based analyses, the Henshin (Arendt et al., 2010) graph-transformation engine was lifted to support product lines of graphs (Salay et al., 2014). These lifted analyses were written from scratch, without reusing any components from their respective product-based analyses. Our approach, on the other hand, lifts an entire class of product-based analyses written as Datalog rules, by lifting their inference engine (and extracting presence conditions together with facts).

SPL^{Lift} (Bodden et al., 2013) extends IFDS (Reps et al., 1995) data-flow analyses to product lines. Model checkers based on Featured Transition Systems (Classen et al., 2013) check temporal properties of transition-system models 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.

Lifting query languages used to implement analyses instead of lifting a single analysis is the approach we are using in this paper. Particularly, we use a variability-aware Datalog engine (Shahin and Chechik, 2020b) that implicitly lifts analyses written in Datalog (Shahin et al., 2019). This approach has also been recently extended to lift analyses written in more expressive, Turing-complete languages (Shahin and Chechik, 2020a).

Variability-aware Visualization. Colour is used effectively in SPL visualization to represent traceability links between source code and feature models. Tools like CIDE (Kästner et al., 2009b), FeatureMapper (Heidenreich et al., 2008), fmp2rsm (Czarnecki and Pietroszek, 2006), and FeatureVISU (Apel and Beyer, 2011) enable colouring of model entities or source-code fragments according to their association with a set of features that the user selects. Our visualization can similarly aid a user in visualizing the results of a variability-aware analysis that apply to a subset of the SPL’s products. Visualization tools that employ interactive techniques, such as detail-on-demand and highlighting, are proven to contribute to the engineer’s comprehension of a product line and to their productivity in modifying the feature configurations (Asadi et al., 2016).

The visualization presented in (Loesch and Ploedereder, 2007) similarly supports an analysis of the feature configuration of a software product line. The authors use Formal Concept Analysis to identify obsolete variable features to optimize the configuration of the product line. Their graph visualization explores the spatial distribution of the nodes and their size to respectively encode

the difference between objects and attributes, and the number of feature variables associated with each node. Our work differs in terms of the goal of the analysis and data encoding provided by the visualization.

Work that is closer to ours include the visualizations provided by VISIT-FC (Botterweck et al., 2008), which support the understanding of possible consequences of the engineer’s decision. The tool provides an interactive view connecting three models (decision, feature, and component models) where the user can select features and visualize the decisions and components related to their selection and the relations between them. The traceability is visualized by explicit links connecting the models’ components. The highlighting of those links is performed by colouring all non-relevant entities in gray, colouring only the information relevant to the engineer.

Recently, Strüber et al. (2020) used graph visualization to represent variability of class diagrams. The authors experimented with three methods to represent variability exploring colour coding and graphical layout. Our visualizer differs by focusing on models extracted from the code with entities and relationships that are more diverse and detailed than class diagrams. Moreover, our visualization focuses on encoding the variability of the models as colour-coded edge groups, not necessarily changing the spatial distribution of nodes.

Our visualization focuses on filtering and highlighting of variability-aware analysis results with respect to how the results apply to particular product sets, rather than filtering and highlighting subsets of a variable system. We recognize that those types of visualization focus on feature- and product-specific aspects of some base representation that exhibits variability, and techniques developed to visualize slices of systems-with-variability could be used to visualize and highlight slices of analysis results-with-variability.

9 Conclusion and Future Work

In this paper, we presented an industrial study of applying a declarative source-code analysis to relational models of annotative Software Product Lines (SPLs). We integrated source-code fact extraction and a variability-aware Datalog engine from two prior projects Shahin et al. (2019); Muscedere et al. (2019), implementing an analysis pipeline. In addition to adapter components between pieces coming from different projects, we enhanced the fact extraction to be variability-aware and added a result-filtering and visualization module for the interactive inspection of results.

We applied the pipeline to a component-interaction behaviour-alteration analysis of models of six automotive controller SPLs from General Motors, each with hundreds of product line features. Our results demonstrate the scalability of our variability-aware analysis approach to real-life industrial SPLs. Analyzing the whole SPL, with presence conditions that relate source-code facts to their associated products, was only 5-59% slower than analyzing the entire factbase including all features with no presence conditions. Our interac-

1 tive visualization module allows users to filter the analysis results for a subset
2 of products, allowing for a finer-grained, product-level inspection of results
3 when needed, e.g., a comparison of analysis results for multiple products and
4 change impact analysis.
5

6 For future work, we plan to integrate our analysis pipeline more tightly to
7 produce a single tool that takes SPL code as input and provides an interactive
8 user interface for inspecting results. We are also in discussions with General
9 Motors to apply the pipeline to other analyses and to more SPLs. In addition,
10 since our pipeline is analysis-agnostic, we are also in the process of identifying
11 other analyses that might be of value to General Motors and whether they
12 can be implemented in Datalog. We also aim to validate the usability of our
13 visualization module by studying its impact on engineer comprehension during
14 the inspection of the analysis results.
15

16 **References**

- 17
18
19
20 Apel S, Beyer D (2011) Feature Cohesion in Software Product Lines: An Ex-
21 ploratory Study. In: Proc. of ICSE'11, ACM, New York, NY, USA, pp 421–
22 430
23 Apel S, Kaestner C (2009) An Overview of Feature-Oriented Software Devel-
24 opment. *J Object Technol* 8:49–84
25 Arendt T, Biermann E, Jurack S, Krause C, Taentzer G (2010) Henshin: Ad-
26 vanced Concepts and Tools for In-place EMF Model Transformations. In:
27 Proc. of MODELS'10, Springer-Verlag, Berlin, Heidelberg, pp 121–135
28 Asadi M, Soltani S, Gašević D, Hatala M (2016) The Effects of Visualiza-
29 tion and Interaction Techniques on Feature Model Configuration. *Empirical*
30 *Software Engineering* 21(4):1706–1743
31 Bodden E, Tolêdo T, Ribeiro M, Brabrand C, Borba P, Mezini M (2013)
32 SPLIFT: Statically Analyzing Software Product Lines in Minutes Instead
33 of Years. In: Proc. of PLDI'13, ACM, pp 355–364
34 Botterweck G, Thiel S, Nestor D, bin Abid S, Cawley C (2008) Visual Tool
35 Support for Configuring and Understanding Software Product Lines. In:
36 Proc. of SPLC'08, IEEE, pp 77–86
37 Ceri S, Gottlob G, Tanca L (1989) What You Always Wanted to Know about
38 Datalog (And Never Dared to Ask). *IEEE Trans on Knowledge and Data*
39 *Engineering* 1(1):146–166
40 Classen A, Heymans P, Schobbens PY, Legay A, Raskin JF (2010) Model
41 Checking Lots of Systems: Efficient Verification of Temporal Properties in
42 Software Product Lines. In: Proc. of ICSE'10, ACM, New York, NY, USA,
43 pp 335–344
44 Classen A, Cordy M, Schobbens PY, Heymans P, Legay A, Raskin JF
45 (2013) Featured Transition Systems: Foundations for Verifying Variability-
46 Intensive Systems and Their Application to LTL Model Checking. *IEEE*
47 *Trans Softw Eng* 39(8):1069–1089
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

- Clements P, Northrop L (2001) *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional
- Czarnecki K, Pietroszek K (2006) Verifying Feature-Based Model Templates against Well-Formedness OCL Constraints. In: *Proc. of GPCE'06*, pp 211–220
- Ernst MD, Badros GJ, Notkin D (2002) An Empirical Analysis of C Preprocessor Use. *IEEE Trans Softw Eng* 28(12):1146–1170
- Gacek C, Anastasopoulos M (2001) Implementing Product Line Variabilities. In: *Proc. of SSR'01*
- Gazzillo P, Grimm R (2012) SuperC: Parsing All of C by Taming the Preprocessor. In: *Proc. of PLDI'12*, ACM, pp 323–334
- Heidenreich F, Şavga I, Wende C (2008) On Controlled Visualisations in Software Product Line Engineering. In: *Proc. of ViSPLE@SPLC'08*
- Holt RC (1997) *An Introduction to TA: The Tuple-Attribute Language*
- Kang K, Cohen S, Hess J, Novak W, Peterson A (1990) *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA
- Kästner C, Apel S, Trujillo S, Kuhlemann M, Batory D (2009a) Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach. In: Oriol M, Meyer B (eds) *Objects, Components, Models and Patterns*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 175–194
- Kästner C, Apel S, Trujillo S, Kuhlemann M, Batory D (2009b) Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach. In: *Proc. of Int. Conf. on Objects, Components, Models and Patterns*, Springer, pp 175–194
- Kästner C, Giarrusso PG, Rendel T, Erdweg S, Ostermann K, Berger T (2011) Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In: *Proc. of OOPSLA'11*, ACM, pp 805–824
- Kästner C, Apel S, Thüm T, Saake G (2012) Type Checking Annotation-based Product Lines. *ACM Trans Softw Eng Methodol* 21(3):14:1–14:39
- Liebig J, Apel S, Lengauer C, Kästner C, Schulze M (2010) An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In: *Proc. of ICSE'10*, ACM, New York, NY, USA, p 105114
- Liebig J, von Rhein A, Kästner C, Apel S, Dörre J, Lengauer C (2013) Scalable Analysis of Variable Software. In: *Proc. of ESEC/FSE'13*, pp 81–91
- Loesch F, Ploedereder E (2007) Optimization of Variability in Software Product Lines. In: *Proc. of SPLC'07*, IEEE, pp 151–162
- Midtgaard J, Dimovski AS, Brabrand C, Wąsowski A (2015) Systematic Derivation of Correct Variability-aware Program Analyses. *Sci Comput Program* 105(C):145–170
- Muscaredere BJ, Hackman R, Anbarnam D, Atlee JM, Davis IJ, Godfrey MW (2019) Detecting Feature-Interaction Symptoms in Automotive Software Using Lightweight Analysis. In: *Proc. of SANER'19*, IEEE

- 1 Reps T, Horwitz S, Sagiv M (1995) Precise Interprocedural Dataflow Analysis
2 via Graph Reachability. In: Proc. of POPL'95, ACM, pp 49–61
- 3 Salay R, Famelis M, Rubin J, Di Sandro A, Chechik M (2014) Lifting Model
4 Transformations to Product Lines. In: Proc. of ICSE'14, ACM, New York,
5 NY, USA, pp 117–128
- 6 Schaefer I, Bettini L, Bono V, Damiani F, Tanzarella N (2010) Delta-Oriented
7 Programming of Software Product Lines. In: Bosch J, Lee J (eds) Proc. of
8 SPLC'10, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 77–91
- 9 Shahin R, Chechik M (2020a) Automatic and Efficient Variability-Aware Lift-
10 ing of Functional Programs. In: Proc. of OOPSLA'20
- 11 Shahin R, Chechik M (2020b) Variability-Aware Datalog. In: Komendantskaya
12 E, Liu YA (eds) Proc. of PADL'20, Springer, LNCS, vol 12007, pp 213–221
- 13 Shahin R, Chechik M, Salay R (2019) Lifting Datalog-based Analyses to Soft-
14 ware Product Lines. In: Proc. of ESEC/FSE'19, ACM, New York, NY, USA,
15 pp 39–49
- 16 Shahin R, Akhundov M, Chechik M (2021a) Software Product Line Analysis
17 Using Variability-aware Datalog. IEEE Transactions on Software Engineer-
18 ing (to appear) URL <https://doi.org/10.36227/techrxiv.14870187.v1>
- 19 Shahin R, Hackman R, Toledo R, Ramesh S, Atlee JM, Chechik M (2021b)
20 Applying Declarative Analysis to Software Product Line Models: An Indus-
21 trial Study. In: 2021 ACM/IEEE 24th International Conference on Model
22 Driven Engineering Languages and Systems (MODELS), pp 145–155, DOI
23 10.1109/MODELS50736.2021.00023
- 24 Somenzi F (1998) CUDD: CU Decision Diagram Package Release 2.2.0
- 25 Strüber D, Anjorin A, Berger T (2020) Variability representations in class
26 models: An empirical assessment. In: Proceedings of the 23rd ACM/IEEE
27 International Conference on Model Driven Engineering Languages and Sys-
28 tems, pp 240–251
- 29 Thüm T, Apel S, Kästner C, Schaefer I, Saake G (2014) A Classification and
30 Survey of Analysis Strategies for Software Product Lines. ACM Comput
31 Surv 47(1):6:1–6:45
- 32 Young B, Cheatwood J, Peterson T, Flores R, Clements P (2017) Product Line
33 Engineering Meets Model Based Engineering in the Defense and Automotive
34 Industries. In: Proc. of SPLC'17, New York, NY, USA, p 175179

Rex
AST
Processing

Tuple-
Attribute
Generation

$FA \wedge !FB \wedge FC$

Filter

Feature expression

Filter

Feature expression

Filter

FA/IFB/FC

Filter

Feature expression

Filter

A

Theme 1 Theme 2

Edge Types

- alterBehaviour
- write
- varWrite
- varInfFunc

Feature Expressions

- $FA \wedge FB \wedge FC \wedge FD$
- $FA \wedge FB \wedge FC \wedge \neg FD$
- $FA \wedge FC$
- $FA \wedge FD$

B

Overview

Node Labels

v1 v2 v3

Relationship Types

v1 v2 v3
alterBehaviour (1) write (1) varWrite (1)
varInfFunc (1)

Filters

$FA \wedge FB \wedge FC \wedge FD$ $FA \wedge FB \wedge FC \wedge \neg FD$ $FA \wedge FC$
 $FA \wedge FD$

Displaying 4 nodes, 4 relationships.

C

Feature expression

Filter

A

Theme 1 Theme 2

Edge Types

alterBehaviour
write
varWrite
varInFunc

Feature Expressions

$FA \wedge FB \wedge FC \wedge FD$
 $FA \wedge FB \wedge FC \wedge FD$
 $FA \wedge FC$
 $FA \wedge FD$

B

Overview

Node labels

* (1) cf function (2) cVariable (2)

Relationship Types

* (1) alterBehaviour (1) write (1) varWrite (1)
varInFunc (1)

Filters

$FA \wedge FB \wedge FC \wedge FD$ $FA \wedge FB \wedge FC \wedge FD$ $FA \wedge FC$
 $FA \wedge FD$

Displaying 4 nodes, 4 relationships.

C

Overview

Node labels

* (4)

cFunction (2)

cVariable (2)

Relationship Types

* (4)

alterBehaviour (1)

varInfFunc (1)

varWrite (1)

write (1)

Displaying 4 nodes, 4 relationships.