



# Applying declarative analysis to industrial automotive software product line models

Ramy Shahin<sup>1</sup> · Rafael Toledo<sup>2</sup> · Robert Hackman<sup>2</sup> · Ramesh S<sup>3</sup> · Joanne M. Atlee<sup>2</sup> · Marsha Chechik<sup>1</sup>

Accepted: 11 January 2023 / Published online: 4 February 2023

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

## Abstract

Program analysis of automotive software has several unique challenges, including that the code base is ultra large, comprising over a hundred million lines of code running on a single vehicle; the code is structured as a software product line (SPL) for managing a *family* of related software products from a common set of artifacts; and the analysis results (despite being numerous and despite being variable) need to be presented to the engineer in a way that is manageable. In previous work, we reported on *lifting* declarative analyses to apply to a software product line, rather than to an individual product variant. This paper reports on milestone results from applying lifted declarative analyses (*behaviour alteration*, *recursion analysis*, *simplifiable global variable analysis*, and two of their variants) to automotive software product lines from General Motors and assessing the scalability of the analyses and the effectiveness of reporting to engineers conditional analysis results (i.e., results conditioned on SPL program variants). We also reflect on some of the lessons learned throughout this project.

**Keywords** Software product lines · Automotive · Declarative analysis · Visualization

## 1 Introduction

For the past five years, researchers from the University of Waterloo, the University of Toronto, and General Motors(GM) have been investigating next-generation software analysis tools with the ultimate goal of being able to perform a system-wide analysis of a vehicle's

---

Communicated by: Sigrid Eldh, Davide Falessi, Burak Turhan

This article belongs to the Topical Collection: *Software Engineering in Practice*

✉ Ramy Shahin  
rshahin@cs.toronto.edu

Extended author information available on the last page of the article.

software. Although a system-wide analysis is not yet possible, we have been working to address some of the unique challenges that arise in analysis of automotive software, such as

- automotive software is multi-threaded and distributed across multiple CPUs;
- the code and execution environments in a single vehicle are heterogeneous with respect to programming languages, language variants, and operating systems;
- the code base is ultra large, comprising over a hundred million lines of code running on a single vehicle;
- the software is structured as a *software product line (SPL)* for managing a family of related software products that are differentiated by their *features*; and
- the analysis results (despite being numerous and variable) need to be presented to the engineer in a way that is manageable.

Of these challenges, General Motors has been particularly interested about the last three: that the tools scale to large software systems (at least a million lines of code (LOC), for now), that the analyses accommodate an SPL's program variants, and that the presentation of analysis results not tax the cognitive load of the engineers.

In previous work (Muscedere et al. 2019), we addressed the challenges of distributed software components and heterogeneity by extracting models of the components and their program elements (e.g., functions, variables, function calls, assignments) and linking these together into a model of the software system. To accommodate variability, we extract a model of the software product line for analysis (Shahin et al. 2021b).

A *software product line (SPL)* supports a family of related products, usually developed together as a common set of mandatory and optional features (Clements and Northrop 2001). A *feature* is the unit of variation, and *products* (also called *configurations* or *variants*) are derived from the SPL by selecting among and integrating features from the SPL's feature set. Our work focuses on *annotative SPLs* (and SPL models), in which program elements and functionality that pertain to specific features or feature combinations are annotated with expressions that represent those features.

Analyzing separately each product of a non-trivial SPL is infeasible because the number of potential products grows exponentially with the size of the SPL's feature set, due to the combinatorial nature of SPL features (Liebig et al. 2013). Instead, several researchers have *lifted* different analyses to be *variability-aware* (Thüm et al. 2014), such that the analysis applies to a product line (as opposed to an individual product) and leverages the commonalities among an SPL's products. Multiple types of analyses have been lifted, including parsing (Gazzillo and Grimm 2012; Kästner et al. 2011), type checking (Kästner et al. 2012), static analyses (Bodden et al. 2013; Midtgaard et al. 2015), model checking (Classen et al. 2010), resulting in significantly faster analyses of the SPL compared to the product-based analyses of all the SPL's products. In previous work, we lifted an entire class of analyses by lifting the Datalog (Ceri et al. 1989b) engine to be variability aware (Shahin et al. 2019; Shahin and Chechik 2020b). As a result, declarative analyses (Bravenboer and Smaragdakis 2009; Benton and Fischer 2007; Dawson et al. 1996; Grech and Smaragdakis 2017) that can be expressed as a set of Datalog rules can be used *as-is* as input to the variability-aware Datalog engine to analyze an SPL model (Shahin et al. 2021a).

Another open problem in SPL-based analyses is how to present analysis results, given that they vary for different sets of products. Most research on SPL visualization focuses on documenting and viewing variability and configuration choices (Kang et al. 1990; Czarnecki and Pietroszek 2006). Additional works use configuration views to facilitate the inspection of consequences of configuration decisions (Botterweck et al. 2008); or visualizations of

variability-analysis results to support variability restructuring and management (Loesch and Ploedereeder 2007). In contrast, the goals of our visualization work are to help the engineer understand and explore the results of an SPL analysis from the perspective of different product sets.

Our work focuses on user-defined declarative program analyses (expressed as Datalog rules) that are mostly variants of data-flow and control-flow analyses. In this paper, we leverage our variability-aware Datalog engine to lift five such analyses and apply them to seven automotive controller product lines provided by General Motors. The paper makes the following contributions: (1) We outline the design of a pipeline for variability-aware analysis of product lines implemented in C/C++. (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. Our evaluation compares the performance of analyzing the whole product line against analyzing a single configuration that includes all features. (3) We describe our interactive visualizer to support the exploration of SPL-analysis results and present the results of a small user study that assesses General Motors engineers' feedback on the visualizer. (4) 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 the earlier work by investigating more program analyses; only one analysis (behaviour alteration) was covered in Shahin et al. (2021b). This paper reports on applying the extended set of analyses to seven SPL controllers from General Motors. The seventh SPL controller is a new subject system not described in Shahin et al. (2021b) that was provided by General Motors to stress test our pipeline: it is significantly larger, has many more features and feature combinations, and includes a middleware component that leads to an order of magnitude more results than analyses of the other controllers. This paper also extends and details our interactive visualization and filtering of SPL analysis results and reports on a small user study that provides engineers' feedback on the effectiveness of the visualization in reporting and exploring results that vary by product sets.

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 SPL analysis results. In Sections 5 and 6, we present our industrial examples and the results of applying our lifted analyses to them, respectively; and in Section 7 we present feedback on our interactive visualizer from a small user study involving General Motors engineers. We discuss lessons learned in Section 8, present related work in Section 9, and conclude in Section 10.

## 2 Background

In this section, we briefly define the concepts we build upon in the rest of the paper. In particular, this includes backgrounds on software product lines, declarative analyses of relational models, and variability-aware analyses that can be applied to a entire product line.

### 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 2001). The unit of variability in an

SPL is a *feature*, where each feature can be either present or absent in each of the SPL's 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.

In an *annotative* SPL (Thüm et al. 2014), feature-specific lines of source code are encapsulated in conditional statements that are guarded (annotated) with feature expressions. The SPL's features are represented as compile-time Boolean constants (called *feature variables*), and *feature expressions* are Boolean expressions over the feature variables. For example, the SPL in Fig. 1 has two features, FA and FB; their values *true* or *false* indicate whether their corresponding features are present or absent, respectively, in a product. Consider the code in Component C1: lines 10-17 are specific to products in which the feature FA is present;

```

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

```

(a) Component C1.

```

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

```

(b) Component C2.

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

line 13 is specific only to products in which features FA and FB are both present; and line 15 is specific only to products in which feature FA is present and feature FB is absent.

A specific value assignment to all of the SPL's feature variables is called a (*feature*) *configuration* and denotes a single product in the SPL. The configuration whose feature variables are all *true* is often referred to as the *150% representation* (Beuche et al. 2016) because this configuration generally does not represent a valid product due to constraints among feature selections.

A set of configurations is succinctly expressed as a *presence condition (PC)*, which is a propositional formula over the feature variables.<sup>1</sup> If  $f$  is a feature variable and if  $pc_1$  and  $pc_2$  are PCs that represent arbitrary sets of configurations in an SPL, then the following are also PCs:

$$\begin{aligned} f &\equiv \text{all configurations in which feature } f \text{ is present} \\ !pc_1 &\equiv \text{all configurations in the SPL not belonging to } pc_1 \\ pc_1 \wedge pc_2 &\equiv \text{the intersection of configurations in both } pc_1 \text{ and } pc_2 \\ pc_1 \vee pc_2 &\equiv \text{the union of configurations in either } pc_1 \text{ or } pc_2 \end{aligned}$$

Thus given the SPL depicted in Fig. 1, the PC {FA} represents two configurations: (1) where FA is present and FB is absent, and (2) where both FA and FB are present. Similarly, the PC {!FA} also represents two configurations.

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 process includes the use of tools to perform a variety of software analyses for bug-finding, metric generation, and performance assessment. In most cases, such tools can be applied only to one software product at a time rather than to an entire SPL. The naïve 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 declarative analyses in Datalog. Datalog (Ceri et al. 1989a) 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). For example, line 1 of Fig. 2 is a rule with a single clause in the body (the `varWrite` clause), and the `transVarWrite` is the head (conclusion) of the rule.

Figure 2 shows a Datalog program (simplified for presentation purposes) for detecting symptoms of *behaviour alteration*, in which a variable assignment in one component affects the behaviour of another component. Lines 1-3 compute the transitive closure of the `varWrite` relationship, thereby finding all data-flows in which one variable is used in the

<sup>1</sup>We use the syntax  $!$ ,  $\wedge$ ,  $\vee$  for the propositional operators *not*, *and*, and *or*, respectively, to be consistent with the syntax of PCs used in our interactive visualization tool (please see Section 4).

```

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

assignment expression of another variable (including parameter assignments). Lines 5-10 define behaviour alteration as a data-flow that starts with a variable assignment (`write`) in function `f0`, which impacts the values of other variables (via `transVarWrite`), and ends with a variable whose value influences the invocation of some function `f1` (`varInfFunc`). As we are interested only in behaviour alterations that cross component boundaries, we exclude intra-component results (lines 8-10).

Running this analysis on facts extracted from the code in Fig. 1 reports 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, indicating that variable `GlobVar` affects whether the function `foo` is invoked (lines 12-13 in `C2`).

### 2.3 Lifted Declarative Analyses

Several software analyses have been re-designed and implemented to enable efficient analysis of the whole SPL at once. Such analyses are called *variability-aware analyses*, and the process of transforming a single-product analysis into an variability-aware analysis is 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  $f$ .

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 effectively lifting any and every product-based analysis that can be expressed in the lifted language. For example, Shahin et al. (2019) lifted Datalog analyses 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<sup>↑</sup> (lifted Soufflé) Datalog engine (Shahin and Chechik 2020b). The result is that analyses written in Datalog are naturally lifted when they are processed by Soufflé<sup>↑</sup>.

Consider a variability-aware version of the behaviour alteration analysis presented in the previous section and consider again the results of running this analysis on the example SPL in Fig. 1; if you recall, the analysis reports that the function `updateX` in component `C1` may influence whether the function `foo` in `C2` is called. The initial `write` relationship from function `updateX` to variable `GlobVar` (line 15 in `C1`) exists only in products that

include feature FA and exclude feature FB (i.e., whose PC is  $FA \wedge !FB$ ). Similarly, the intermediate `varWrite` relationship from the variable `GlobalVar` to itself (line 15 in C1) exists only in products with the PC  $FA \wedge !FB$ . The `varInFunc` relationship that ends the data-flow (lines 12-13 in C2) exists in all products. The full data-flow path result exists only in products that satisfy the *conjunction* of the PCs of all the edges in the path: that is, all products in the PC  $FA \wedge !FB$ . In general, a variability-aware analysis is expected to report its results annotated with the products (or PC) for which each result applies.

### 3 The Analysis Pipeline

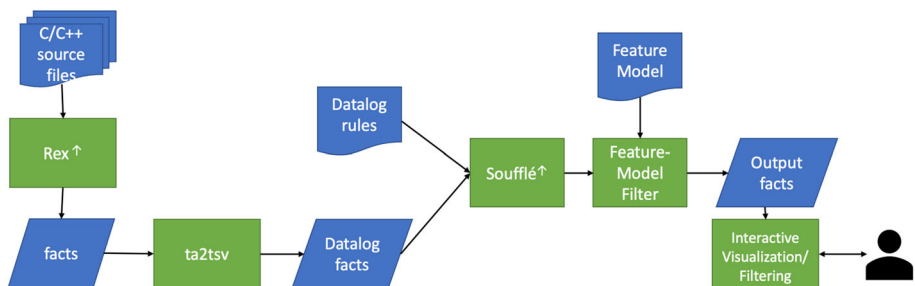
We have implemented an end-to-end pipeline for extracting a product line model from source code, analyzing it, and interactively visualizing the results. The analysis pipeline integrates components used in previous projects (Shahin et al. 2019; Muscedere et al. 2019), together with some adapter components for converting data from one format to another. The overall pipeline design is shown in Fig. 3. The major tools of this pipeline are `Rex↑`, `Soufflé↑`, and the Interactive Visualization/Filtering tool.

An SPL model is extracted from C/C++ source files using a new variability-aware version of `Rex` (Muscedere et al. 2019), 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 as tuples and then converted to Datalog fact format using a simple script (`ta2tsv` adapter component).

`Soufflé↑` takes as input facts annotated with presence conditions and infers additional facts based on a set of input Datalog rules that express analyses of interest. `Soufflé↑`'s output is presented as an annotated graph, in which each presented result is annotated with a presence condition denoting the set of products for which the result applies. The engineer can then use the Interactive Visualizer to create filters that highlight (with colour) the analysis results that apply particular product sets of interest. The visualization component is explained in more detail in Section 4.

#### 3.1 Variability-Aware Fact Extraction

Our analyses operate on extracted *facts* about C/C++ source code, rather than operating on the code itself, to improve the scalability of our analyses to large software systems.



**Fig. 3** End-to-end fact extraction and analysis pipeline. Source code is provided to `Rex↑`, which produces a factbase of program facts that `ta2tsv` translates into the input expected by the SPL-analyzer `Soufflé↑`. An analysis, expressed as Datalog rules, is input to `Soufflé↑` along with these facts, and the analysis results are presented to the user via an interactive visualizer

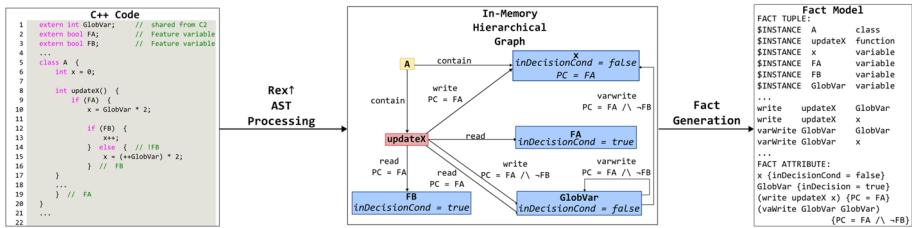


Fig. 4 The fact extraction of component C1

Specifically, a fact extractor Rex (Muscedere et al. 2019), based on the *Clang++* open-source compiler,<sup>2</sup> parses C/C++ source-code files, generates abstract syntax trees (ASTs), and extracts facts of interest from the AST into an in-memory hierarchical graph. Source-code entities such as variable declarations and function declarations are the nodes of the graph; and relations such as variable assignments (in which one variable is used in 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 *fact model* or *factbase*) about source-code entities, their relations, and their respective attributes represented as three-tuples (triples).

In order to support analysis of SPL models, we developed a variability-aware version of Rex<sup>↑</sup> that annotates entities and relationships with their presence conditions. A Rex<sup>↑</sup> user can specify, by type and naming convention, which program variables are to be considered feature variables and thus used in presence conditions (e.g., only constant global `bool` or `enum` type variables). Variability-aware Rex<sup>↑</sup> keeps track of all conditions over feature variables that hold while walking the AST and uses that information to annotate facts with their PCs as they are extracted.

Figure 4 gives an overview of the Rex<sup>↑</sup> extraction process of the component C1 in Fig. 1a. On the left is the input, in this case C++ code; the middle of the figure depicts extracted information as an in-memory hierarchical graph; and on the right is the extracted fact model in tuple format. In this example, Rex<sup>↑</sup> creates fact nodes for the class `A`, function `updateX`, and variables `x`, `FA`, `FB`, and `GlobalVar`.<sup>3</sup> Each `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 `GlobalVar` in an assignment to variable `x` in C1), a `varWrite` edge is created from the used variable to the assigned variable (e.g., `varWrite GlobalVar x`). The creation of the other edges follows the same pattern. Attributes of entities and relationships are listed at the end of the fact 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 feature variables. Thus, variability-aware Rex<sup>↑</sup> extracts a 150% representation that includes facts for all the SPL’s features, where conditional facts

<sup>2</sup><https://clang.lvm.org/index.html>

<sup>3</sup>The names of the entities are simplified for this example to improve legibility. In practice, Rex creates long identifier names that capture the entity’s context (i.e., enclosing function, class, etc., up to and including filename).



are annotated with their products' presence conditions. 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).

The fact model is translated into the input format of the Soufflé<sup>†</sup> reasoner, using a script *ta2tsv* that we wrote specifically for that purpose. For example, in a fact model, presence conditions are listed as attributes at the end of the file rather than being co-located with their associated facts. Our *ta2tsv* script associates each fact with its corresponding presence-condition attribute.

### 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 devised lifted versions of these analyses by expressing each as a set of Datalog rules and “executing” them using the lifted Datalog engine Soufflé<sup>†</sup>. This way we were able to leverage the flexibility of using Datalog as a query language for expressing analyses. We were also able to leverage all the optimizations in Soufflé<sup>†</sup> to help ensure that our analyses scale to industrial-size SPLs.

#### 3.2.1 Behaviour Alteration Analysis

In our work, the primary analyses of interest are those that detect possible *component interactions*, where a component behaves differently in isolation versus when it is combined with other components (Muscedere et al. 2019). Such analyses are of particular interest to General Motors because of the large number of components and component combinations in their products and product lines. An engineering team will know its components well, but will not necessarily know all of the ways in which its components can affect the behaviours of components developed by other teams. One of the most complex types of component interaction is *behaviour alteration* (Muscedere et al. 2019), a form of data-flow component interaction, in which a change to a variable value made in one component alters the behaviour of another component. The specific instance of behaviour alteration used in this paper is (1) an assignment is made in component *C1* to a variable *v*; (2) whose modified value impacts other variables through variable assignments and impacts other components through parameter passing; until (3) in another component *Cn* a variable *x*, whose value has been impacted by the modified value of *v*, is used in the decision condition of some control structure (i.e., an *if*, *for*, *while*, or *switch* statement) that (4) guards a function call. Thus, the analysis looks for a data flow from a variable assignment in one component to a control structure in another component, where the control-structure's statement block includes a function call. Figure 1 gives a simple example where the write to *globVar* in line 15 of component *C1* could affect whether or not the function *bar* calls the function *foo* in component *C2*.

To analyze the software controllers provided by General Motors, we developed a specialized version of the behaviour 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

A second analysis of interest to General Motors involves recursion. Automotive software typically uses little to no recursion in order to guarantee timing requirements. Our work initially focused on two types of recursion:

- (1) function recursion
- (2) component recursion

The first analysis detects functions that directly or indirectly call themselves via a cycle of function calls within a single component; and the second analysis detects a cycle of function calls involving functions from at least two distinct components. As we discuss in Section 6, results reported by the recursion analysis led GM engineers to request us to perform a followup analysis to help them understand the instances and contexts of the reported occurrences of recursion. This followup analysis is discussed in Section 8.4.

The recursion analyses exemplify simple coding standards, like MISRA C<sup>4</sup> standards that are used in the automotive and other safety-critical industries. These kinds of code patterns can be expressed easily as Datalog queries.

### 3.2.3 Simplifiable Global Variable Analysis

The third analysis of interest detects a code pattern that was of particular interest to a General Motors engineer for potential code refactorings. Specifically, 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 refactored as a parameter of the function that reads from it – which is useful because global variables can introduce unnecessary couplings of components and potential logical errors in maintaining their state.

Figure 5 illustrates a simplifiable global variable `CtrlIdx`, where `X` and `Z` are the only functions in the program that call 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. To facilitate this, we developed an interactive visualizer that enables the engineer to apply coloured filters to the results to help identify groups of paths occurring in related software products.

---

<sup>4</sup><https://www.misra.org.uk/>

```

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`

Our visualizer is implemented on top of the Neo4j Browser,<sup>5</sup> the user interface provided by the open-source graph database Neo4j.<sup>6</sup> As a database engine, Neo4j enables the storing and querying of graphical data like the facts and results of our analyses. As such, we import our results into an instance of the Neo4j database to be queried and visualized.

Figure 6 shows a pedagogical example of analysis results comprising functions ( $f1$ ,  $f2$ ), variables ( $v1$ ,  $v2$ ), their relationships, and their respective presence conditions over feature variables ( $FA$ ,  $FB$ ,  $FC$ ,  $FD$ ) in a visualization frame. Each visualization frame has a central interactive area in which the graphical results are displayed; the user can use native Neo4j Browser facilities to zoom in and out, and rearrange the layout of the graph. The visualization frame also includes an expandable sidebar that gives an overview of the data being represented and provides options to customize the appearance of elements, including node size, edge thickness, and the information presented on the labels. For each edge in the graph, the visualizer displays both of the type of the edge and the edge's presence condition. The edge type appears in bold whereas the presence condition is located on the opposite side of the edge. 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 (please see Fig. 7). In the textbox on the top right, the engineer specifies a filter as a presence condition representing a set of SPL configurations, and the visualizer highlights the subset of results that satisfy the filter's presence condition. Our visualizer employs Logic Solver,<sup>7</sup> a Boolean satisfiability solver, to reason for each fact whether the fact's presence condition satisfies also each 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.

<sup>5</sup><https://neo4j.com/developer/neo4j-browser/>

<sup>6</sup><https://neo4j.com/>

<sup>7</sup><https://github.com/meteor/logic-solver>

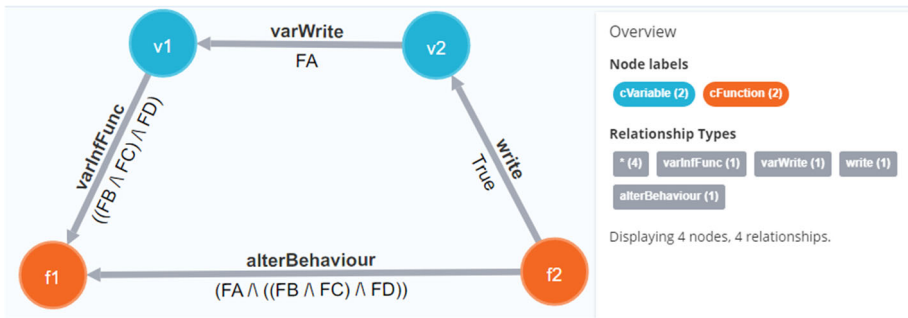


Fig. 6 Neo4j Browser interface

Multiple filters can be applied to the same analysis results, producing a colour-coded graph visualization that highlights which analysis results pertain to specific configurations. For example, as shown in Fig. 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  $FA \wedge FB \wedge FC \wedge FD$ . After applying filters, the engineer can inspect the legend at bottom right corner showing the filters applied and their colours (shown in Fig. 7-B).

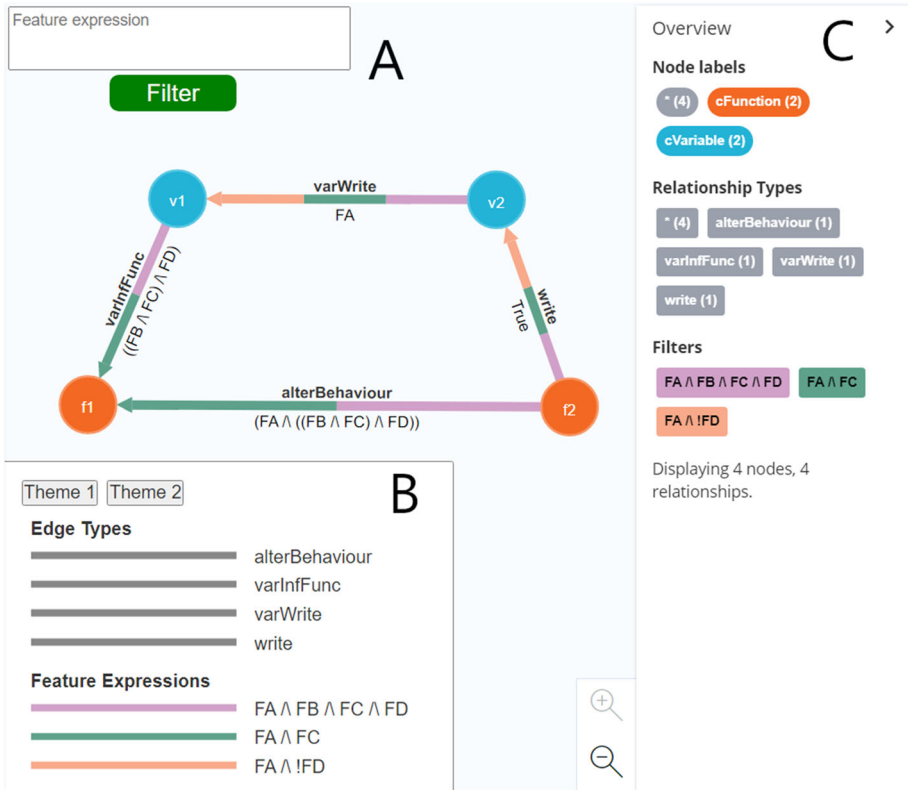
Through the application of one or more filters, the engineer can explore and better understand the analysis results. A single filter can be used to determine and visualize which results apply to a particular set of software products of interest. Alternatively, the engineer can compare how two or more sets of software products differ in their analysis results by applying multiple filters, one for each product set. Moreover, the engineer 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 coloured yellow highlight the analysis results that satisfy only the first filter and the edges coloured blue and yellow highlight the analysis results that satisfy both filters.<sup>8</sup>

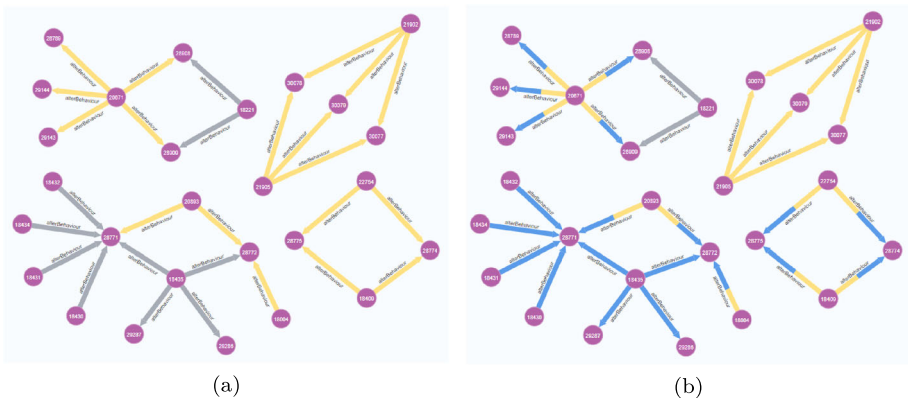
Visualization of numerous analysis results is always a concern (Von Landesberger et al. 2011), but there are several facilities within the tool chain for searching and scoping the presented results. Firstly, the analysis itself (i.e., the Datalog query) can be used to limit the number of results returned and in some cases can prioritize results (e.g., prioritize shortest or longest path results, or prioritize results that involve the largest number of components). The analysis query can also be refined to focus only on specific components. Secondly, the engineer's web browser's text-search feature can be used to search the graph labels to localize results related to particular components, functions, or variables. Thirdly, native Neo4j Browser features can be used to include or exclude node or edge types from the visualization, to more easily focus on results that pertain to nodes and edges of interest.<sup>9</sup>

<sup>8</sup>Node labels and presence conditions on edges have been omitted from Fig. 8(b) to avoid revealing proprietary information.

<sup>9</sup>Neo4j Browser manual. <https://neo4j.com/docs/browser-manual/current/>



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



**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 initial (yellow) filter

**Table 1** Size metrics for the seven product lines analyzed

|            | 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 |

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)

Finally, our extension to Neo4j Browser can be used to highlight the analysis results that pertain to product sets of interest. In Section 7, we report on a small user study in which GM engineers evaluate our interactive visualizer for the tasks viewing, searching, highlighting, and comparing analysis results for different product sets.

## 5 Industrial SPL Examples

To assess scalability, we applied our analyses of interest to SPL 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. Of particular interest is controller SPL-G, which provided a stress test of our tool chain. SPL-G is significantly larger than the other controllers, has many more features and feature combinations, and includes a middleware component that leads to an order of magnitude more results than analyses of the other controllers.

The General Motors controllers encode the inclusion or exclusion of features using what are called *configuration parameters*. Configuration parameters are represented as global constants of enumerated types (enum) or Boolean type (bool).<sup>10</sup> 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.

For some of our analyses (specifically, behaviour alteration analysis and component recursion analysis), 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

<sup>10</sup>Configuration parameters are *feature variables*, which were described in Section 2.1.

facts allowed us to adapt our analyses to avoid reporting intra-component results (please see the last line in Fig. 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-world 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 Rex<sup>↑</sup> to automatically extract both 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 Datalog facts. For each of the analyses, we applied Soufflé (version 1.3.1) to the 150% representation and applied Soufflé<sup>↑</sup> to the factbase annotated with presence conditions. To assess performance, we repeated each analysis experiment five times and removed the minimum and maximum execution times (to marginalize the effect of noise from the execution environment) and report the mean-average execution times and standard deviations for both the 150% representation and the product line. Tables 2, 3, 4, 5 and 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**), where the percentage of variational facts to the total number of facts is **VFacts (%)**. The total number of feature variables referenced by the presence conditions of the variational facts is **Features** (approximated to the

**Table 2** Results of applying the behaviour alteration analysis

|                         | SPL-A  | SPL-B  | SPL-C  | SPL-D  | SPL-E  | SPL-F  | SPL-G   |
|-------------------------|--------|--------|--------|--------|--------|--------|---------|
| <b>Features</b>         | ~400   | ~500   | ~900   | ~600   | ~600   | ~500   | ~1600   |
| <b>Facts</b>            | 157303 | 225538 | 215120 | 228185 | 227241 | 226640 | 621714  |
| <b>VFacts</b>           | 698    | 1070   | 2126   | 1148   | 2078   | 955    | 3944    |
| <b>VFacts (%)</b>       | 0.44   | 0.47   | 0.99   | 0.50   | 0.91   | 0.42   | 0.63    |
| <b>150% Time (sec.)</b> | 3.93   | 5.14   | 17.62  | 4.81   | 17.76  | 4.87   | 175.17  |
| <b>(Std. Dev.)</b>      | 0.01   | 0.02   | 0.04   | 0.06   | 0.01   | 0.06   | 2.46    |
| <b>150% Outputs</b>     | 128780 | 177806 | 399085 | 167243 | 438319 | 177689 | 1594370 |
| <b>150% Results</b>     | 1191   | 1527   | 2757   | 1725   | 3887   | 1400   | 22832   |
| <b>Distinct PCs</b>     | 198    | 278    | 779    | 318    | 1123   | 278    | 2225    |
| <b>Time (sec.)</b>      | 6.49   | 8.29   | 28.68  | 7.79   | 32.63  | 7.62   | 427.60  |
| <b>(Std. Dev.)</b>      | 0.09   | 0.04   | 0.37   | 0.08   | 1.15   | 0.01   | 8.24    |
| <b>Outputs</b>          | 128759 | 177801 | 399027 | 167241 | 436740 | 177684 | 1592336 |
| <b>VOutputs</b>         | 505    | 676    | 3484   | 756    | 19046  | 924    | 127279  |
| <b>VOutputs (%)</b>     | 0.39%  | 0.38%  | 0.87%  | 0.45%  | 4.36%  | 0.52%  | 7.99%   |
| <b>Results</b>          | 1189   | 1525   | 2757   | 1724   | 3881   | 1398   | 22822   |
| <b>Overhead</b>         | 64.82% | 61.41% | 62.71% | 61.95% | 83.68% | 56.46% | 144.11% |

**Table 3** Results of applying the GM variant behaviour alteration analysis

|                         | SPL-A  | SPL-B  | SPL-C  | SPL-D  | SPL-E  | SPL-F  | SPL-G   |
|-------------------------|--------|--------|--------|--------|--------|--------|---------|
| <b>Features</b>         | ~400   | ~500   | ~900   | ~600   | ~600   | ~500   | ~1600   |
| <b>Facts</b>            | 173454 | 241689 | 231271 | 244336 | 243392 | 242791 | 637865  |
| <b>VFacts</b>           | 698    | 1070   | 2126   | 1148   | 2078   | 955    | 3944    |
| <b>VFacts (%)</b>       | 0.40   | 0.44   | 0.92   | 0.47   | 0.85   | 0.39   | 0.62    |
| <b>150% Time (sec.)</b> | 3.99   | 5.17   | 17.72  | 4.83   | 17.84  | 4.88   | 173.06  |
| <b>(Std. Dev.)</b>      | 0.00   | 0.03   | 0.11   | 0.02   | 0.07   | 0.03   | 0.20    |
| <b>150% Outputs</b>     | 142457 | 191403 | 411397 | 180496 | 454691 | 191423 | 1607902 |
| <b>150% Results</b>     | 15     | 23     | 109    | 23     | 421    | 23     | 802     |
| <b>Distinct PCs</b>     | 197    | 276    | 771    | 317    | 1111   | 276    | 2213    |
| <b>Time (sec.)</b>      | 6.57   | 8.30   | 28.60  | 7.85   | 32.05  | 7.60   | 410.06  |
| <b>(Std. Dev.)</b>      | 0.00   | 0.10   | 0.09   | 0.10   | 0.19   | 0.04   | 2.93    |
| <b>Outputs</b>          | 142436 | 191398 | 411339 | 180494 | 453095 | 191418 | 1605848 |
| <b>VOutputs</b>         | 527    | 690    | 3253   | 768    | 18796  | 938    | 127522  |
| <b>VOutputs (%)</b>     | 0.37%  | 0.36%  | 0.79%  | 0.43%  | 4.15%  | 0.49%  | 7.94%   |
| <b>Results</b>          | 15     | 23     | 109    | 23     | 421    | 23     | 802     |
| <b>Overhead</b>         | 64.77% | 60.72% | 61.34% | 62.43% | 79.69% | 55.59% | 136.94% |

nearest hundreds). For example, when applying the behaviour alteration analysis (Table 2) to SPL-A, the number of input facts relevant to behaviour alteration is 157303, 698 of which are variational, corresponding to 0.44% of the relevant input facts.

**Table 4** Results of applying the simplifiable global variable analysis

|                         | SPL-A   | SPL-B   | SPL-C   | SPL-D   | SPL-E   | SPL-F   | SPL-G  |
|-------------------------|---------|---------|---------|---------|---------|---------|--------|
| <b>Features</b>         | ~400    | ~500    | ~900    | ~600    | ~600    | ~500    | ~1800  |
| <b>Facts</b>            | 228473  | 333016  | 285853  | 334635  | 303877  | 338233  | 769236 |
| <b>VFacts</b>           | 710     | 1009    | 1624    | 1080    | 1693    | 879     | 3160   |
| <b>VFacts (%)</b>       | 0.31    | 0.30    | 0.57    | 0.32    | 0.56    | 0.26    | 0.41   |
| <b>150% Time (sec.)</b> | 4780.88 | 9377.31 | 4672.73 | 9526.3  | 5859.57 | 9615.21 | 25967  |
| <b>(Std. Dev.)</b>      | 15.99   | 62.43   | 20.54   | 52.00   | 51.22   | 160.96  | 382.88 |
| <b>150% Outputs</b>     | 129629  | 178061  | 145877  | 177084  | 151796  | 177988  | 412600 |
| <b>150% Results</b>     | 89      | 117     | 108     | 106     | 160     | 97      | 459    |
| <b>Distinct PCs</b>     | 245     | 355     | 755     | 392     | 557     | 341     | 2328   |
| <b>Time (sec.)</b>      | 5102.92 | 10032.1 | 5090.34 | 10225.2 | 6280.87 | 10381.3 | 28618  |
| <b>(Std. Dev.)</b>      | 128.82  | 207.34  | 95.24   | 188.95  | 25.66   | 327.02  | 485.86 |
| <b>Outputs</b>          | 129649  | 178079  | 145904  | 177110  | 151840  | 178005  | 412688 |
| <b>VOutputs</b>         | 1226    | 1556    | 1580    | 1598    | 1833    | 1493    | 3790   |
| <b>VOutputs (%)</b>     | 0.95%   | 0.87%   | 1.08%   | 0.90%   | 1.21%   | 0.84%   | 0.92%  |
| <b>Results</b>          | 89      | 117     | 108     | 106     | 160     | 97      | 459    |
| <b>Overhead</b>         | 6.74%   | 6.98%   | 8.94%   | 7.34%   | 7.19%   | 7.97%   | 10.21% |



**Table 5** Results of applying the function recursion analysis

|                         | SPL-A  | SPL-B  | SPL-C  | SPL-D  | SPL-E  | SPL-F  | SPL-G   |
|-------------------------|--------|--------|--------|--------|--------|--------|---------|
| <b>Features</b>         | ~300   | ~300   | ~600   | ~300   | ~400   | ~300   | ~1200   |
| <b>Facts</b>            | 51264  | 72154  | 65373  | 74314  | 80767  | 73521  | 163629  |
| <b>VFacts</b>           | 292    | 420    | 756    | 426    | 569    | 356    | 1097    |
| <b>VFacts (%)</b>       | 0.57   | 0.58   | 1.16   | 0.57   | 0.70   | 0.48   | 0.67    |
| <b>150% Time (sec.)</b> | 1.46   | 1.93   | 2.00   | 2.01   | 2.72   | 1.84   | 6.56    |
| <b>(Std. Dev.)</b>      | 0.01   | 0.02   | 0.02   | 0.02   | 0.02   | 0.00   | 0.11    |
| <b>150% Outputs</b>     | 285229 | 347712 | 358868 | 364255 | 502085 | 332209 | 1244323 |
| <b>150% Results</b>     | 4      | 4      | 0      | 4      | 6      | 4      | 10      |
| <b>Distinct PCs</b>     | 199    | 264    | 1158   | 264    | 360    | 247    | 793     |
| <b>Time (sec.)</b>      | 2.17   | 2.74   | 3.39   | 2.94   | 3.78   | 2.57   | 23.26   |
| <b>(Std. Dev.)</b>      | 0.06   | 0.08   | 0.13   | 0.13   | 0.10   | 0.08   | 1.25    |
| <b>Outputs</b>          | 285229 | 347712 | 358868 | 364255 | 502085 | 332209 | 1244323 |
| <b>VOutputs</b>         | 3533   | 6960   | 5503   | 7223   | 6224   | 4328   | 14119   |
| <b>VOutputs (%)</b>     | 1.24%  | 2.00%  | 1.53%  | 1.98%  | 1.24%  | 1.30%  | 1.13%   |
| <b>Results</b>          | 4      | 4      | 0      | 4      | 6      | 4      | 10      |
| <b>Overhead</b>         | 48.44% | 42.22% | 69.44% | 46.10% | 38.93% | 40.18% | 254.36% |

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 Tables 2–6) in each of the controllers.

**Table 6** Results of the component recursion analysis

|                         | SPL-A  | SPL-B  | SPL-C  | SPL-D  | SPL-E  | SPL-F  | SPL-G   |
|-------------------------|--------|--------|--------|--------|--------|--------|---------|
| <b>Features</b>         | ~300   | ~300   | ~600   | ~300   | ~400   | ~300   | ~1200   |
| <b>Facts</b>            | 111509 | 158174 | 117422 | 162635 | 142584 | 162281 | 309441  |
| <b>VFacts</b>           | 292    | 420    | 954    | 426    | 715    | 356    | 1097    |
| <b>VFacts (%)</b>       | 0.26   | 0.27   | 0.81   | 0.26   | 0.50   | 0.22   | 0.35    |
| <b>150% Time (sec.)</b> | 1.60   | 2.09   | 2.19   | 2.17   | 3.40   | 2.02   | 7.86    |
| <b>(Std. Dev.)</b>      | 0.00   | 0.00   | 0.00   | 0.00   | 0.03   | 0.00   | 0.05    |
| <b>150% Outputs</b>     | 315726 | 382327 | 424390 | 398987 | 681000 | 369020 | 1565191 |
| <b>150% Results</b>     | 0      | 0      | 0      | 0      | 0      | 0      | 0       |
| <b>Distinct PCs</b>     | 140    | 191    | 776    | 192    | 354    | 175    | 643     |
| <b>Time (sec.)</b>      | 2.29   | 2.95   | 3.60   | 3.01   | 4.35   | 2.73   | 22.78   |
| <b>(Std. Dev.)</b>      | 0.08   | 0.08   | 0.14   | 0.07   | 0.01   | 0.04   | 1.75    |
| <b>Outputs</b>          | 315726 | 382327 | 424390 | 398987 | 680999 | 369020 | 1565191 |
| <b>VOutputs</b>         | 3694   | 7811   | 11033  | 8106   | 17204  | 4556   | 17234   |
| <b>VOutputs (%)</b>     | 1.17%  | 2.04%  | 2.60%  | 2.03%  | 2.53%  | 1.23%  | 1.10%   |
| <b>Results</b>          | 0      | 0      | 0      | 0      | 0      | 0      | 0       |
| <b>Overhead</b>         | 42.57% | 40.99% | 64.11% | 38.42% | 27.89% | 34.81% | 189.91% |

Because the number of possible products is exponential in the number of configuration parameters, the large number of configuration parameters makes analyzing individual product variants infeasible.

As reported in Table 2, the behaviour alteration analysis of the 150% representation of SPL-A using Soufflé takes 3.93 seconds, and the number of inferred facts (**150% Outputs**) is 128780. Among those, 1191 facts are the end results of the analysis (**150% Results**). However, applying the same analysis to the variational facts of the same SPL-A product line using Soufflé<sup>†</sup> takes 6.49 seconds (an overhead of only 64.82%). Soufflé<sup>†</sup> infers 128759 new facts (**Outputs**), 505 of which are variational (**VOutputs**). Among the outputs, 1189 are end results of the analysis (**Results**). The number of distinct presence conditions calculated as part of the analysis is 198.

Considering all our analyses, variational analysis time overheads range from 6.74% (simplified global variable analysis applied to SPL-A) to 254.36% (recursion checking applied 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 simplified global variable analysis, the overheads are significantly lower than those of the other, lighter-weight, analyses – in part because this analysis applied the SPLs' 150% representation models are so costly. 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 inferred by the analyses, including all intermediate facts generated as part of the computation of final results. There are two reasons behind the discrepancy in the number of facts inferred when analyzing the 150% representation versus applying Soufflé<sup>†</sup> 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, whereas a non-variational aggregator always generates a single result.

We also measured the total number of unique presence conditions (**Distinct PCs**) computed during the inference process.<sup>11</sup> 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

---

<sup>11</sup>This measurement was aided by the fact that the presence conditions are stored as Binary Decision Diagrams (BDDs) (Bryant 1992), and BDDs have canonical representations.

shows that variability-aware analysis scales to large-scale industrial software product lines with hundreds of features.

## 7 GM Engineers' Feedback on Graph Visualization

Graph visualization tools can include a number of features to help engineers cope with a large amount of data (see Section 4). In this section, we report on a semi-formal user survey with General Motors engineers to help evaluate the effectiveness of an extension we introduced to the Neo4J browser to use in graph visualization to inspect conditional (product-specific) analysis results. Specifically, we asked for General Motors engineers' feedback on three aspects of our work: (1) the capacity of associating variability-aware analyses with product sets, (2) the preferred way to represent results of variability-aware analyses, and (3) the utility of coloured filters to help engineers explore and focus on subsets of results.

### 7.1 Methodology

We began by delivering an online presentation that showed the output of variability-aware analyses (including presence condition annotations), the graphical and tabular representation of such results, and the expression of filters to highlight subsets of results that pertain to specific product sets. We created a video of this same presentation for those engineers who were unable to attend the presentation. We also provided access to a prototype of our interactive interface so that engineers could experiment with the interactive graph visualization and its coloured filters.

The survey comprised three sections, each asking a set of questions about the participant's preferences using a 5-point Likert scale (ranging from a strong preference to a strong dislike) followed by opportunities to provide free-form answers about the rationale behind their preferences.

- The first section focuses on the participant's interest in seeing analysis results annotated with software variants and contained four questions (including two optional open-ended questions) which assessed the degree to which the association of analysis results with product sets is useful in understanding the results.
- The second section asks participants about their preference between graphical and tabular formats. A tabular format that lists results (e.g., program elements or paths that match some pattern of interest), each annotated with a presence-condition attribute, is more conventional. A graphical format presents the same information with graph edges annotated with their respective presence condition. This section of the survey comprises seventeen questions asking about the participants' general preference (e.g. "*Overall, which format do you prefer?*") and task-specific preferences (e.g., reading, searching, understanding). At the end of the section, the respondents were asked to rank how the presented task-specific scenarios influenced their general preferences.
- The third section of the survey gauges the participants' interest in using coloured filters to highlight analysis results. The coloured filters apply to both tabular and graphical representations, highlighting subsets of results based on user-specified product sets of interest. Similarly to the previous section, the survey asks seventeen questions about the participants' general preference (e.g., "*Overall, which visualization (coloured or uncoloured) would be faster and easier to read, understand, find, and report results*").

*and their associated configuration expressions?”)* and task-specific preferences (e.g., *“to access the impact of analysis results from adding or removing a feature from a configuration expression”*). At the end of the section, participants could also rank the presented task-specific scenarios with respect to their general preferences.

- The survey concludes with two open-ended questions asking participants about the general impression of the presented features of the tool and their suggestions for improvement.

Six senior software engineers from General Motors responded to our survey. The complete set of materials used in this study (e.g., slides, video, survey questions, responses) are publicly available.<sup>12</sup>

## 7.2 Answers: Associating Variability-Aware Analyses Results with Product Sets

All of the respondents reported a preference for associating analysis results with a *set* of products. Most respondents (5 out of 6) considered it useful to know which analysis results belong to a *single* product. One of the participants explained how the association between results and product sets could help with understanding the results:

*“Configurable software operates in many different ways which is fundamentally hard to keep clear in doing software tasks, knowing when code is active/inactive is a key understanding element ... [products set] knowledge allows you to understand the shared interactions, I cannot change something as unique to variants if shared across variants”* (P1).

The observed clear preference for understanding sets of results justifies and guides our efforts for building visualizations of variability-aware analysis results.

## 7.3 Answers: Presentation of Variability-Aware Analysis Results

The respondents expressed clear but diverse preferences for graphical versus tabular results. One participant (P5) prefers the graphical representation for most of the task-specific scenarios presented in the survey. Three participants (P2, P3, P4) prefer the opposite, favouring the tabular format for the same set of tasks. The other two respondents (P1, P6) prefer different representations for different tasks and showed interest in having an interface that provides both formats. For example, P6 said they would like a graphical view of results to *read* data values and a tabular view to *find* particular results.

Two participants who favour the tabular format admitted to preferring the graphical format (or not having a preference) if the data could be further queried to focus on a subset of the results. Such comments reflect a concern (shared by most participants) that a graph representation would become difficult to read once it becomes large enough. For the participants who prefer the graphical format, the graph helps direct their focus when inspecting the results:

*“I really prefer them both together, the graph provides an easier pattern perspective of the whole but can be hard to consume details. I would see myself using the graph to find nodes then the table to dig in on detail”* (P1).

<sup>12</sup><https://github.com/toledorafael/emse2022userSurvey>

*“The graphical format allows me to quickly zoom in to the part of the dataflow I want to analyze further, without having to stop and mentally connect the individual call links” (P5).*

The received answers provide sufficient evidence for the need to provide support for both tabular and graphical formats. Further investigation of how the two representations could cooperate to deliver the best experience to engineers is left for future work.

## 7.4 Answers: Utility of Coloured Highlighting of Filtered Analysis Results

Most of the respondents indicated that coloured filters are strongly helpful for the overall understanding of analysis results. They said that the use cases that most benefitted from the coloured filters were: (a) identifying results associated with a single product instance, (b) assessing how a change in software configuration impacts the analysis results, and (c) comparing facts associated with different product sets. P5 explained that the coloured filters could be especially helpful in understanding the impact of configuration decisions:

*“[...]where I really need to do detailed analysis is on product instances within that set (e.g., if I set one value within a product set a specific way, exactly which subset of the original filtered set of possible paths is impacted?). This seems like the most useful case for understanding behaviour, or whether my software design has any gaps in the conditions I have set up, where I expect a common set of dataflows to be highlighted for every one [product] of a product set, but for an individual product instance within that collection there is a difference.”*

Another respondent suggested that presence-condition annotations should include the number of variants in the presence condition’s product set, and we intend to implement this suggestion in our tool.

Participant (P2) asserted that coloured filters would not be helpful to them because they are colour-blind: in fact, colour choices could make it hard for them to read and distinguish between results. This particular respondent may have been influenced by the survey’s use of specific colours (red, blue, yellow) when showing images of highlighted (filtered) results, whereas in practice, the choice of filter colours is controllable by the user. In any case, the participant’s response shows a limitation of this feature for those engineers who experience full colour blindness.

To summarize, the use of coloured filters to highlight subsets of analysis results looks promising but should be confirmed by further study. Respondents’ feedback classify coloured filters as a useful feature but also point out potential improvements and limitations that should be considered for an optimal experience.

## 7.5 Threats to Validity

The main threat to validity of our results is the small number of engineers who participated in our user study. We shared the survey with twelve engineers, and they were invited to share the materials with other engineers whom they thought would be interested and appropriate. We received six responses to our survey, all from senior engineers with 20 to 30 years of work experience as software engineers and 15 to 30 years of experience working with configurable software. We believe that the respondents’ seniority and expertise lend significant weight to their answers, even if the number of respondents is not enough for study results to be statistically significant. We are currently working on our next user study that aims to

reach a broader group of engineers. Our evaluation is decidedly semi-formal and qualitative rather than quantitative. Yet we were able to collect preliminary but consistent results that the visualization tooling we are building is effective.

## 8 Lessons Learned

In this section, we reflect on some of the lessons learned by conducting these empirical studies.

### 8.1 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.

### 8.2 Variability Encoding

Soufflé<sup>†</sup> 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 operators (e.g., addition, 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  $x < \text{Feat2}$  is a logically valid presence condition, but is not acceptable in Soufflé<sup>†</sup>. We apply a syntactic transformation for these kinds of expressions, turning the above expression into a Boolean variable  $x\_LT\_Feat2$ , where the `_LT_` sub-string stands for less-than. We use similar substitutions for other comparison operators. These transformations are applied by a post-processing script that is executed on the facts before they are added to the factbase. The transformations are limited to comparison operators (not arithmetic or bitwise operators) and the expressions to which a feature variable is compared is limited to enum constants.

The fact that the four features belonging to `FeatSet` are mutually exclusive can be expressed as a constraint on feature variables in the feature model of the product line. The fragment of the feature model representing this property for `FeatSet` is:

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

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

$$(Feat0 \vee Feat1 \vee Feat2 \vee Feat3)$$

### 8.3 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<sup>†</sup> in earlier work (Shahin et al. 2019) is marginal, but that was reported for relatively small benchmarks of only tens of features each. Results presented in Shahin et al. (2021b) show that the performance overhead of full product line analysis using Soufflé<sup>†</sup> is still marginal for industrial product lines, with hundreds of features. Evaluation results in Section 6 reinforce those findings with a more diverse set of analyses and one more (significantly bigger in size) benchmark from General Motors.

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.

#### 8.4 Utility of Analysis Results in Practice

Automated analyses help engineers identify underlying facts about the program. For example, General Motors engineers were surprised by the results of the recursion analysis and were interested in tracking down the causes and occurrences of some of them. While recursive code is not necessarily prohibited in automotive software, detected instances are worthy of inspection as they can be resource intensive (with respect to memory and time). The followup analyses focused on recursion results detected in SPL-G, in part because we could collaborate with a General Motors engineer familiar with this controller. The followup analyses reported more detailed results including the software components involved, the function names, and code snippets. For SPL-G, these analyses detected three functions that directly call themselves (direct recursion) and four pairs of functions that mutually call each other (indirect recursion). Upon examining these results, the engineer was able to determine that: (1) all instances of direct recursion and two instances of mutual recursion belong to a component dedicated to testing the system code and thus the recursion was deemed not harmful to the well-functioning of the system; and (2) the two other instances of mutual recursion belong to the system code, but their constituent function calls cannot occur together because they are guarded by mutually exclusive conditions on program variables.

Ultimately, our recursive analyses did not identify any problematic instances of recursion (at least in SPL-G). However, the original analysis and its followups were still deemed useful: (1) Because our analyses are expressed in a query language, we were able to specialize both the followup analyses and level of detail reported in the analysis results; and (2) the engineer increased their confidence in the code.

### 9 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 graph-transformation engine (Arendt et al. 2010) 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 the inference engine (and inferring presence conditions together with facts).

SPL<sup>Lift</sup> (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 with 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.

In this paper, we use a lifted query language to implement analyses instead of lifting existing analyses. In particular, 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** Our work on visualization differs from previous works in SPL visualization in both the type of information that is visualized and the user's ability to filter and highlight information.

The conventional use of colour in SPL visualization is to distinguish features or variabilities in source code or 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. 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 analysis of the feature configurations of a software product line. The authors use Formal Concept Analysis to identify and remove obsolete variable features to optimize the task of configuring the product line. Their graph visualization explores the spatial distribution of the nodes (representing concepts) and the node sizes, to encode the difference between them and the number of feature variables associated with each node, respectively. The authors use black, white, and gray colours to indicate the number of features attached to each node and identify obsolete features. Our work differs in terms of the goal of the analysis and data presented in the visualization.

Work that is closer to ours are the visualizations provided by VISIT-FC (Botterweck et al. 2008) to support the understanding of possible consequences of the engineer's decisions. 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. Their analyses reflect consequences of decisions about a single configuration whereas our analyses visualize results for sets of products.

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 program elements 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.

In summary, previous works use colour or tags to associate portions of an SPL's code or models with distinct features to highlight the SPL's variabilities, or they visualize the consequences of feature selections on the product configuration. Whereas our visualizer uses colour to highlight subsets of analysis results that belong to *sets of products* specified by the engineer: the engineer defines one or more product sets of interest and the visualizer

highlights the corresponding analysis results by painting each result with all the colours representing all of the product sets to which the result belongs. Hence, our visualizer supports exploration, filtering, highlighting, and comparing of analysis results rather than simple presentation of the results.

## 10 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 five analyses (*behaviour alteration*, *recursion analysis*, *simplifiable global variable analysis*, and two of their variants) of models of seven 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. Our interactive visualization module allows users to filter the analysis results for a subset of products, allowing for a finer-grained inspection of results per project or per project set (e.g., enabling comparison of analysis results for different feature selections and change-impact analysis).

With respect to limitations, (1) our analyses need to be declarative and expressible in Datalog. So far, this has not posed a limitation on the types of analyses we have tried to perform. (2) Our variability-aware analysis incurs a performance overhead of 6.74%-254.36% to analyze the entire SPL, compared to the time to analyze the superset of all the SPL's features. However, we consider this overhead to be negligible, given that the analysis returns results for all of the SPL's products; the runtime of a brute-force approach that applies the same analysis to each product separately would grow exponentially with the number of the SPL features. (3) The use of colour to highlight analysis results may be a limitation for users who suffer from full colour blindness. (4) The visualization of analysis results suffers when the set of results is very large. A small semi-formal user study of GM engineers provides preliminary evidence that filtering and highlighting can help the engineer to focus their attention on subsets of results of interest, but these findings and the impact of other facilities provided by the visualization environment need to be confirmed with larger studies.

For future work, we plan to integrate our analysis pipeline more tightly to produce a single tool that takes SPL code as input and provides an interactive user interface for inspecting results. We are also in discussions with General Motors to apply the pipeline to other analyses and to more SPLs. In addition, since our pipeline is analysis-agnostic, we are also in the process of identifying other analyses that might be of value to General Motors and whether they can be implemented in Datalog.

**Data Availability** Data sharing not applicable to this article as no datasets were generated or analyzed during the current study.

## Declarations

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

## References


- Apel S, Beyer D (2011) Feature cohesion in software product lines: an exploratory study. In: Proc. of ICSE'11. ACM, New York, pp 421–430
- Apel S, Kästner C (2009) An overview of feature-oriented software development. *J Object Technol* 8:49–84
- Arendt T, Biermann E, Jurack S, Krause C, Taentzer G (2010) Henshin: advanced concepts and tools for in-place EMF model transformations. In: Proc. of MODELS'10. Springer-Verlag, Berlin, pp 121–135
- Asadi M, Soltani S, Gašević D, Hatala M (2016) The effects of visualization and interaction techniques on feature model configuration. *Empir Softw Eng* 21(4):1706–1743
- Benton WC, Fischer CN (2007) Interactive, Scalable, Declarative Program Analysis: From Prototype to Implementation. In: Proc. of PPDP'07. ACM, New York, pp 13–24
- Beuche D, Schulze M, Duvigneau M (2016) When 150% is too much: supporting product centric viewpoints in an industrial product line. In: Proceedings of the 20th international systems and software product line conference, SPLC '16. Association for Computing Machinery, New York, pp 262–269
- Bodden E, Tolêdo T, Ribeiro M, Brabrand C, Borba P, Mezini M (2013) SPLIFT: statically analyzing software product lines in minutes instead of years. In: Proc. of PLDI'13. ACM, pp 355–364
- Botterweck G, Thiel S, Nestor D, Bin Abid S, Cawley C (2008) Visual tool support for configuring and understanding software product lines. In: Proc. of SPLC'08. IEEE, pp 77–86
- Bravenboer M, Smaragdakis Y (2009) Strictly declarative specification of sophisticated points-to analyses. In: Proc. of OOPSLA'09. ACM, New York, pp 243–262
- Bryant RE (1992) Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput Surv* 24(3):293–318
- Ceri S, Gottlob G, Tanca L (1989a) What you always wanted to know about Datalog (And Never Dared to Ask). *IEEE Trans Knowl Data Eng* 1(1):146–166
- Ceri S, Gottlob G, Tanca L et al (1989b) What you always wanted to know about Datalog (And Never Dared to Ask). *IEEE Trans Knowl Data Eng* 1(1):146–166
- Classen A, Heymans P, Schobbens PY, Legay A, Raskin JF (2010) Model checking lots of systems: efficient verification of temporal properties in software product lines. In: Proc. of ICSE'10. ACM, New York, pp 335–344
- Classen A, Cordy M, Schobbens PY, Heymans P, Legay A, Raskin JF (2013) Featured transition systems: foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans Softw Eng* 39(8):1069–1089
- Clements P, Northrop L (2001) Software product lines: practices and patterns. Addison-Wesley Professional, Reading
- Czarniecki K, Pietroszek K (2006) Verifying feature-based model templates against well-formedness OCL constraints. In: Proc. of GPCE'06, pp 211–220
- Dawson S, Ramakrishnan CR, Warrenm DS (1996) Practical program analysis using general purpose logic programming systems: a case study. In: Proc. of PLDI'96. ACM, New York, pp 117–126
- 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
- Grech N, Smaragdakis Y (2017) P/Taint: Unified points-to and taint analysis. *Proc ACM Program Lang* 1:1–28
- Heidenreich F, Şavga I, Wende C (2008) On controlled visualisations in software product line engineering. In: Proc. of ViSPLE@SPLC'08, pp 335–341
- 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, 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, pp 105–114
- 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
- Muscudere BJ, Hackman R, Anbarnam D, Atlee JM, Davis JJ, Godfrey MW (2019) Detecting feature-interaction symptoms in automotive software using lightweight analysis. In: *Proc. of SANER'19*. IEEE, pp 175–185
- Reps T, Horwitz S, Sagiv M (1995) Precise interprocedural dataflow analysis via graph reachability. In: *Proc. of POPL'95*. ACM, pp 49–61
- Salay R, Famelis M, Rubin J, Di Sandro A, Chechik M (2014) Lifting model transformations to product lines. In: *Proc. of ICSE'14*. ACM, New York, pp 117–128
- Schaefer I, Bettini L, Bono V, Damiani F, Tanzarella N (2010) Delta-oriented programming of software product lines. In: Bosch J, Lee J (eds) *Proc. of SPLC'10*. Springer, Berlin, pp 77–91
- Shahin R, Chechik M (2020a) Automatic and efficient variability-aware lifting of functional programs. In: *Proc. of OOPSLA'20*, pp 1–27
- Shahin R, Chechik M (2020b) Variability-aware datalog. In: Komendantskaya E, Liu YA (eds) *Proc. of PADL'20*, LNCS, vol 12007. Springer, pp 213–221
- Shahin R, Chechik M, Salay R (2019) Lifting datalog-based analyses to software product lines. In: *Proc. of ESEC/FSE'19*. ACM, New York, pp 39–49
- Shahin R, Akhundov M, Chechik M (2021a) Software Product Line Analysis Using Variability-aware Datalog. *IEEE Transactions on Software Engineering* (to appear). <https://doi.org/10.36227/techrxiv.14870187.v1>
- Shahin R, Hackman R, Toledo R, Ramesh S, Atlee JM, Chechik M (2021b) Applying declarative analysis to software product line models: an industrial study. In: 2021 ACM/IEEE 24th international conference on model driven engineering languages and systems (MODELS), pp 145–155. <https://doi.org/10.1109/MODELS50736.2021.00023>
- Strüber D, Anjorin A, Berger T (2020) Variability representations in class models: an empirical assessment. In: *Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems*, pp 240–251
- Thüm T, Apel S, Kästner C, Schaefer I, Saake G (2014) A classification and survey of analysis strategies for software product lines. *ACM Comput Surv* 47(1):6:1–6:45
- Von Landesberger T, Kuijper A, Schreck T, Kohlhammer J, van Wijk JJ, Fekete JD, Fellner DW (2011) Visual analysis of large graphs: state-of-the-art and future research challenges. In: *Computer graph forum*, Wiley Online Library, vol 30, pp 1719–1749
- Young B, Cheatwood J, Peterson T, Flores R, Clements P (2017) Product line engineering meets model based engineering in the defense and automotive industries. In: *Proc. of SPLC'17*, New York, pp 175–179

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

## Affiliations

Ramy Shahin<sup>1</sup>  · Rafael Toledo<sup>2</sup> · Robert Hackman<sup>2</sup> · Ramesh S<sup>3</sup> ·  
Joanne M. Atlee<sup>2</sup> · Marsha Chechik<sup>1</sup>

Rafael Toledo  
rftoledo@uwaterloo.ca

Robert Hackman  
r2hackma@uwaterloo.ca

Ramesh S  
ramesh.s@gm.com

Joanne M. Atlee  
jmatlee@uwaterloo.ca

Marsha Chechik  
chechik@cs.toronto.edu

<sup>1</sup> University of Toronto, Toronto, ON, Canada

<sup>2</sup> University of Waterloo, Waterloo, ON, Canada

<sup>3</sup> General Motors, Warren, MI, USA