Variability-aware Neo4j for Analyzing a Graphical Model of a Software Product Line

by

Xiang Chen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

© Xiang Chen 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

A *Software product line (SPLs)* eases the development of families of related products by managing and integrating a collection of mandatory and optional *features* (units of functionality). Individual products can be derived from the product line by selecting among the optional features. Companies that successfully employ SPLs report dramatic improvements in rapid product development, software quality, labour needs, support for mass customization, and time to market.

In a product line of reasonable size, it is impractical to verify every product because the number of possible feature combinations is exponential in the number of features. As a result, developers might verify a small fraction of products and limit the choices offered to consumers, thereby foregoing one of the greatest promises of product lines — mass customization.

To improve the efficiency of analyzing SPLs, (1) we analyze a *model* of an SPL rather than its code and (2) we analyze the SPL model itself rather than models of its products. We extract a model comprising *facts* (e.g., functions, variables, assignments) from an SPL's source-code artifacts. The facts from different software components are linked together into a lightweight model of the code, called a *factbase*. The resulting factbase is a typed graphical model that can be analyzed using the Neo4j graph database.

In this thesis, we *lift* the Neo4j query engine to reason over a factbase of an entire SPL. By lifting the Neo4j query engine, we enable *any* analysis that can be expressed in the query language to be applicable to an SPL model. The lifted analyses return *variability-aware results*, in which each result is annotated with a feature expression denoting the products to which the result applies.

We evaluated lifted $\widehat{Neo4j}$ on five real-world open-source SPLs, with respect to ten commonly used analyses of interest. The first evaluation aims at comparing the performance of a post-processing approach versus an on-the-fly approach computing the feature expressions that annotate to variability-aware results of $\widehat{Neo4j}$. In general, the on-the-fly approach has a smaller runtime compared to the post-processing approach. The second evaluation aims at assessing the overhead of analyzing a model of an SPL versus a model of a single product, which ranges from 1.88% to 456%. In the third evaluation, we compare the outputs and performance of $\widehat{Neo4j}$ to a related work that employs the variability-aware *V-Soufflé* Datalog engine. We found that $\widehat{Neo4j}$ is usually more efficient than *V-Soufflé* when returning the same results (i.e., the end points of path results). When $\widehat{Neo4j}$ returns complete path results, it is generally slower than *V-Soufflé*, although $\widehat{Neo4j}$ can outperform *V-Soufflé* on analyses that return short fixed-length paths.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Joanne M. Atlee, for accepting me as a master student in her research group. I am truly thankful for her exceptional mentorship, expertise, and invaluable insights that have guided me throughout my study. Her support, encouragement, and constructive feedback have been instrumental in shaping this work. I will always be grateful to Jo for her patience and help.

I am also grateful for being part of such a warm and welcoming research group, and I would like to express my appreciation to my colleagues for their continuous willingness to provide help and support.

I also want to thank my mom and dad. Without their unwavering love and support, I would not have been able to complete this work. They have been my guiding light, encouraging me to persevere and believe in myself throughout this journey.

Special thanks to Professor Nancy A. Day and Professor Michael W. Godfrey for their invaluable constructive feedback as my thesis readers.

Last but not least, I also want to thank my adorable cats, Lion and Momo, and my beloved dog Anthony for being my constant companions. I am grateful for their silent support, which reminded me to take breaks and cherish the simple joys of life.

Dedication

To my family. I will always be grateful for your love and support.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Thesis Statement 5

1.2 Thesis Contributions 6

1.3 Thesis Organization 6

2 Background 8

2.1 Software Product Lines 8

2.2 Software Factbases and $\widehat{factbase}$ 9

2.3 Neo4j and Graph Database 12

2.4 Variability-aware *V-Soufflé* 16

2.5 Summary 18

3 Variability-Aware $\widehat{\text{Neo4j}}$ 20

3.1 Traversal Framework 21
3.2 Lifted $\widehat{Factbase}$ 22
3.3 Lifted Traversal Algorithm 23
3.4 Soundness 28
3.5 Summary 29

4 Analyses of interest 30

4.1 Direct and Indirect Inter-Component-Based Communication 31
4.2 Loop Detection 32
4.3 Behaviour Alternation 33
4.4 Multiple Callers 33
4.5 Race Condition 34
4.6 Direct and Indirect Recursion 35
4.7 Call Graph Analysis 36
4.8 Triangle-Shaped Communication Patterns 36
4.9 Summary 37

5 Evaluation 38

5.1 Experimental Setup 38
5.2 Post-processing Versus On-the-fly Approach 41
5.3 Scalability of $\widehat{Neo4j}$ 44
5.4 $\widehat{Neo4j}$ versus *V-Soufflé* 47
5.5 Implications and lessons learned 51
5.6 Threats to validity 51
5.7 Summary 52

6 Related Work 54

7 Conclusion **56**

7.1 Limitations 56

7.2 Future Work 57

References **59**

List of Figures

1.1 Product-based analysis toolchain . . . 2
1.2 Difference of applying varAssign to configuration $FA \wedge FB$ and $\widehat{varAssign}$ to the example SPL . . . 5
2.1 $\widehat{factbase}$ for the SPL in Listing 1.1 . . . 11
2.2 Neo4j Browser interface . . . 14
2.3 Enhancements to visualize variability-aware analysis results of $\widehat{varAssign}$ on Listing 1.1 . . . 15
2.4 Analysis pipeline using *V-Soufflé* [59] . . . 16
3.1 Structure of Our Toolchain . . . 20
3.2 $\widehat{Neo4j}$: Modifications and Additions to Neo4j . . . 21
3.3 A traversal example . . . 25
4.1 Direct inter-component-based communication pattern . . . 31
4.2 Indirect inter-component-based communication pattern . . . 31
4.3 Loop detection pattern . . . 32
4.4 Behaviour alternation pattern . . . 33
4.5 Multiple callers pattern . . . 34
4.6 Race condition pattern . . . 34
4.7 Direct recursion pattern . . . 35
4.8 Indirect recursion pattern . . . 35

4.9 Call graph analysis pattern 36
4.10 Triangle-shaped communication patterns 36

5.1 Original macro definition and modifications 39
5.2 Evaluation 1 41
5.3 Triangle-shaped communication patterns of AxTLS 50

List of Tables

2.1 Entities and relationships of interests 10

3.1 Iterations of Algorithm 1, performing the second subquery from Listing 3.1 on the $\widehat{factbase}$ from Fig. 2.1 27

5.1 SPLs used in our evaluation 40

5.2 Comparison of on-the-fly vs. post-processing of PCs on $\widehat{Neo4j}$ 42

5.3 Evaluation 1: Average number of results and average runtime per analysis query 43

5.4 Evaluation 1: Average number of results and average runtime per program 43

5.5 $\widehat{Neo4j}$ variability-aware analysis vs. Neo4j product-based analysis 44

5.6 Evaluation 2: Average number of results and average runtime per analysis query 45

5.7 Evaluation 2: Average number of results and average runtime per program 46

5.8 Performance of $\widehat{Neo4j}$ vs. *V-Soufflé* 48

5.9 Evaluation 3: Average number of results and average runtime per analysis query 49

5.10 Evaluation 3: Average number of results and average runtime per program 50

Chapter 1

Introduction

Software variants have become increasingly prevalent in numerous industries, including healthcare, telecommunications, and automotive manufacturing. Imagine, for instance, an automobile manufacturer that produces a variety of vehicle models, each with its own set of features and configurations. These vehicle models can vary in terms of engine specifications, entertainment systems, and safety features. Developing separate software solutions for every model would be inefficient and result in redundant work. To address this challenge, *software product lines* (SPLs) are utilized to manage the variations across these variants effectively. Developers can identify commonalities and variabilities across the variants and then create software solutions as a set of common artifacts.

Companies that successfully employ SPLs report dramatic improvements in rapid product development, software quality, labour needs, support for mass customization, and time to market [67]. Successful stories include an SPL of operational flight programs from Boeing [5], an SPL of engine-control software for gasoline systems from Bosch [5], and the control software for the powertrains from General Motors [5]. These success cases demonstrate that SPL software development has realized the potential to support the efficient development of product variants and mass customization in software engineering.

A *software product line (SPL)* refers to a collection of software systems, known as *products*, that share a common set of required and optional features and are developed using a shared set of artifacts [19]. The central concept within an SPL is the *feature*. Each feature represents a distinct unit of functionality or behaviour that contributes to the overall capabilities of the software. Features serve as the primary source of variability within the product line, allowing different products to be customized by including or excluding specific features according to user requirements. The presence or absence of these features

in a particular product is what distinguishes one product from another, forming its unique *feature configuration*.

This thesis focuses mainly on *annotative product lines*, which use annotative methods to associate source code fragments with features. Typically, these feature annotations take the form of either C preprocessor (CPP) directives that protect feature code or *feature variables* (implemented as global constants) that are used in conditional statements that guard feature code. A condition over feature variables is known as a *presence condition (PC)*, which identifies a product or set of products defined by the presence and absence of the PC's corresponding features. Developers use PCs to guard code associated with the PCs' products.

In a product line of reasonable size, it is impractical to verify every derivable product individually (called *product-based analysis*) because the number of possible feature combinations is exponential in the number of features. As a result, developers might verify only a small fraction of products and limit the choices of products offered to consumers, thereby foregoing one of the greatest assets of product lines — the promise of mass customization [66].

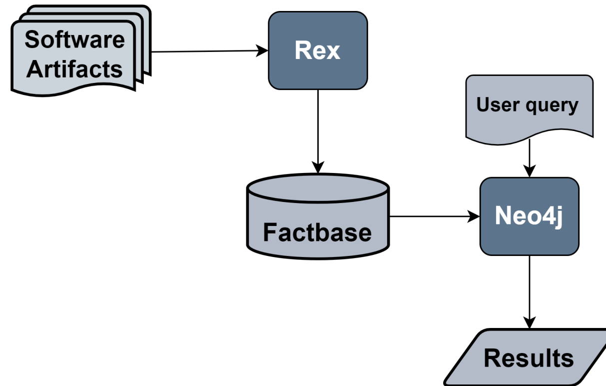


Figure 1.1: Product-based analysis toolchain

In contrast, a *variability-aware analysis* evaluates the whole SPL as a single artifact, ideally leveraging the commonalities across products [62]. Specifically, a variability-aware analysis applies to multiple product variants simultaneously, maximizing the utilization of computed intermediate results. When a product-based analysis is adopted to be a variability-aware analysis, we say it is *lifted* to apply to SPLs. Several product-based analysis techniques have been *lifted* to corresponding variability-aware analyses, such as program analyses [11, 43, 56], parsing [36], model checking [18, 8], and type checking [35]. In this thesis, lifted analyses and lifted artifacts (like SPL models, or the inputs and outputs

```
1  bool FA; // Feature variable
2  bool FB; // Feature variable
3  ...
4  class Example {
5       int a, b, c, d, e, f;
6  ...
7       void changeValue() {
8         if (FA) {
9           d++;        // FA
10        } else {
11          e = d - 1; // !FA
12          a++; // !FA
13        }
14
15        if (FB) {
16          b = a + 1; // FB
17          c = b + 1; // FB
18        } else {
19          e++; // !FB
20        }
21
22        if (c >= 0) { assignC(); }
23        if (e >= 0) { assignE(); }
24      }
25
26      void assignC() { c = f; }
27      void assignE() { e = f; }
28 };
```

Listing 1.1: An SPL containing two features: FA and FB

of lifted analyses) are annotated with a wide hat (e.g., $\widehat{factbase}$) to distinguish them from the original program-based artifacts or analyses.

Fig. 1.1 shows the toolchain for a product-based analysis of a model of a software program [45]. The software source code is input to *Rex* and *Rex* extracts facts and generates a factbase. *Rex* [45] is a C/C++ fact extractor developed by our research team. Example facts of interest include program entities (e.g., variables and functions), relations between entities (e.g., function calls), and attributes on entities or relationships (e.g., whether a function is a callback function). The collection of these pertinent facts is called a *factbase*, which constitutes a graphical model of the software program. We input the factbase to a graph database *Neo4j* [46] along with user queries for analysis. The results returned by

Neo4j can be as simple as nodes or may be paths or subgraphs.

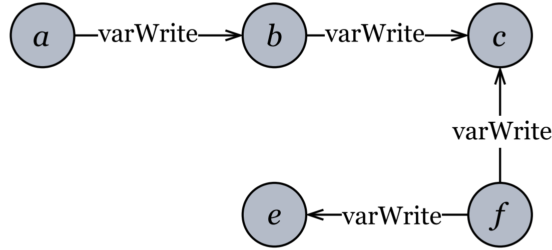
This thesis adapts this toolchain to apply to models of software product lines. Consider a simple example of an SPL written in the C language, as shown in Listing 1.1. This SPL contains two features, FA and FB, which are represented as global Boolean feature variables. The feature variables are used in conditions that guard code fragments associated with their respective features. For example, the lines of code highlighted in red represent an example product with the configuration that includes the presence condition of both features FA and FB.

Extracting a $\widehat{factbase}$ for this SPL means that each fact in the $\widehat{factbase}$ is annotated with its presence condition, which indicates the collection of products in which the fact holds. Variability-aware *Rex* is capable of extracting facts of an SPL and generating a $\widehat{factbase}$. The main contribution of this thesis is to lift the $\widehat{Neo4j}$ engine to enable querying and reasoning over a $\widehat{factbase}$ and generating variability-aware results, which could be nodes, paths, or subgraphs. Each analysis result is annotated with a PC, which is the conjunction of the PCs annotating the result's constituent nodes and edges. If a result's PC is unsatisfiable, then the result does not hold in any of the SPL's products and the result is discarded.

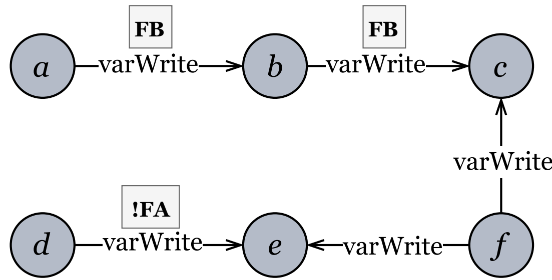
Assume that we want to detect the dataflow of a variable assignment to subsequent assignments, identifying variables whose values are impacted by the initial assignment; let *varAssign* be that analysis that accepts as input a simple program model and returns this dataflow result. This analysis can be applied directly to a product to produce the expected dataflow results. However, when presented with an SPL, the analysis would ideally return the dataflow results for all product variants in an efficient manner.

A lifted version of algorithm $\widehat{varAssign}$ would return a collection of dataflow results, where each result is annotated with a PC representing the products for which the result holds. Fig. 1.2a shows the result of applying *varAssign* to the product whose configuration is $FA \wedge FB$, and Fig. 1.2b shows the result of applying $\widehat{varAssign}$ to the entire example SPL. In Fig. 1.2b, results are annotated with the presence condition FB, $!FA$, or an empty PC. A result annotated with an empty PC means that the result holds in all the products. Similarly, a result annotated with FB means the result holds only in products that include feature FB, and a result annotated with $!FA$ means the result holds only in the products that exclude feature FA.

Shahin et al. [59] assembled a comprehensive toolchain that extracts an SPL model from an SPL code base, analyzes the model, and visualizes the results. First, variability-aware $\widehat{Rex}$ is applied to an SPL's source code for fact extraction, generating a $\widehat{factbase}$.



(a) Applying varAssign to configuration $FA \wedge FB$



(b) Applying $\widehat{varAssign}$ to the example SPL

Figure 1.2: Difference of applying varAssign to configuration $FA \wedge FB$ and $\widehat{varAssign}$ to the example SPL

They lifted a Datalog engine [32] to reason over an SPL $\widehat{factbase}$, which is then imported to the *V-Soufflé* Datalog engine along with a set of Datalog rules (i.e., analyses of interest) to perform variability-aware analysis. The analysis results are then input into the Neo4j database for visualization. One of the weaknesses of Datalog analysis on a graphical model is that the Datalog engine cannot return path results. Instead, the start and end nodes of paths are reported.

1.1 Thesis Statement

This thesis streamlines this toolchain by lifting the Neo4j reasoning engine to enable queries over an SPL $\widehat{factbase}$. By lifting the Neo4j query engine, we improved the efficiency of the lifted $\widehat{Neo4j}$ applied to an SPL model versus applying Neo4j on all of the SPL's product models. Moreover, compared to Shahin et al.'s work, our work has the added benefit of returning richer query results (i.e., paths rather than just endpoints of path results) and

is generally more efficient.

1.2 Thesis Contributions

The following are the contributions made in this thesis:

- We present $\widehat{Neo4j}$, which is a lifted version of the Neo4j query engine that processes queries over the $\widehat{factbase}$ of an SPL and returns variability-aware results.
- We compare the performances of two approaches to variability-aware $\widehat{Neo4j}$ queries: a post-processing approach and an on-the-fly approach. The post-processing approach uses Neo4j as-is to find all query results in the $\widehat{factbase}$, *ignoring* the PCs on facts, and subsequently computes the PCs for all results and filters out results whose PCs are unsatisfiable. The on-the-fly approach computes each result's PC as the result is being constructed.
- We evaluate the efficiency of $\widehat{Neo4j}$ and show that a lifted analysis, which returns results for *all* of an SPL's products, is more efficient than product-based analysis using Neo4j. Specifically, this evaluation compares the performance of variability-aware analyses with that of product-based analyses on a single product (the *maximal product*, comprising all optional features).
- We also compare the performance of the $\widehat{Neo4j}$ query engine with its closest related work: the *V-Soufflé* lifted Datalog engine [59]. Our studies show that $\widehat{Neo4j}$ is usually more efficient than *V-Soufflé* when returning the same results (i.e., the end points of path results), and has the potential to return richer results (i.e., full path results) and to visualize variability-aware results.

1.3 Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2 provides background on SPLs, extracted factbase models, Neo4j, and *V-Soufflé*.

- Chapter 3 details our approach to lifting the Neo4j query engine to operate on an SPL $\widehat{factbase}$ model and return variability-aware query results.
- Chapter 4 explains details of the ten analyses we conducted to evaluate the efficacy and efficiency of our variability-aware analysis.
- Chapter 5 discuss three separate evaluations that we conducted to assess our work: (1) on-the-fly approach versus post-processing approach, (2) scalability of $\widehat{Neo4j}$, and (3) $\widehat{Neo4j}$ versus *V-Soufflé*.
- Chapter 6 provides an overview of the related works in graphical models, graph databases, and lifting product-based analysis to the whole SPL.
- Chapter 7 summarizes the work conducted in this thesis and outlines the limitations, and proposes potential future research that can expand upon the results of this thesis.

Chapter 2

Background

In this chapter, we overview fundamental concepts of software product lines (SPLs), fact extractors for creating models of SPLs, the Neo4j graph database for querying such models, and related work *V-Soufflé* [59].

2.1 Software Product Lines

A *software product line (SPL)* is a family of software systems (called *products*) that share a set of required and optional features and that are developed from a common set of artifacts [19]. A *feature* is the central unit of variability in an SPL, such that each product of an SPL is distinguished by which features are present or absent. The set of features that define a product is commonly referred to as the *feature configuration* of the product.

There are three main categories for engineering SPLs: annotative product lines, compositional product lines, and delta-oriented product lines. In *compositional product lines*, each feature is designed and implemented separately; and in *delta-oriented product lines*, a *core module* represents the fully defined base product and the deltas (the variants) are created based on their deviations from the core product. This thesis focuses mainly on *annotative product lines*. In *annotative* approaches to SPL engineering, the code that implements a specific feature is annotated as being associated with that feature. Such feature annotations generally take the form of either C preprocessor (CPP) directives [34] that guard feature code or *feature variables* used in conditional statements that guard feature code [62]. In the latter approach, a feature variable is a global variable[1] whose value

[1] A feature variable is normally a global constant of type Boolean but can be an enumerated or integer

indicates whether the corresponding feature is present or absent in a particular product. A condition over feature variables is called a *presence condition* (PC), which designates a product or set of products defined by the presence and absence of the PCs corresponding features; PC's are used in the program conditions to guard code associated with the PCs' product(s). Our work is restricted to PC-annotative static SPLs, in which feature selection (i.e., feature variable values) is determined at build time and does not change during execution.

Example. Listing 1.1 presents an SPL that has two features, *FA* and *FB*, represented by global Boolean feature variables. The feature variables are used in program conditions that guard code fragments associated with their respective features. For example, line 9 occurs in products containing feature *FA* and lines 11 and 12 occur in products in which feature *FA* is absent. The lines of code highlighted in red represent an example product with the configuration $FA \wedge FB$, and the lines of code in black are not included in this product.

The number of products in an SPL can be exponential in the number of its features. Thus, separately analyzing every derived product (called *product-based analysis*) may be infeasible [62]. In contrast, a *variability-aware analysis* evaluates the whole SPL as a single artifact, ideally leveraging the commonalities across products [62]. A number of product-based analysis techniques have been *lifted* to corresponding variability-aware analyses that apply to an entire SPL, including program analyses [11, 43, 56], parsing [36], model checking [18, 8], and type checking [35]. In this paper, lifted artifacts and analyses are annotated with a wide hat (e.g., $\widehat{factbase}$) to distinguish them from the original program-based artifacts or analyses.

2.2 Software Factbases and $\widehat{factbase}$

Program analyses can be highly complex and may not be feasible on a large software system or when the source code is distributed or heterogeneous. In such cases, researchers seek to analyze a *model* of the software.

Fact extractors (e.g., ASX [22], CPPX [21], Doop [13], Rex [45]) are analyzers that extract *program facts* from software artifacts, resulting in a collection of pertinent data called a *factbase*. Example facts of interest include program entities (e.g., variables, functions, components), relations between entities (e.g., function calls, variable assignments), and attributes on entities or relationships (e.g., whether a function is a callback function).

type.

Facts can be extracted from source-code artifacts (in different languages or language variants) [45], software design models [29], configuration or build code [39], etc., and combined into a single factbase model of a software system.

In particular, *Rex* [45] is a C/C++ fact extractor, modified from *Clang++*[2], that was developed by our research team and that parses code into an Abstract Syntax Tree (AST) and then walks the AST extracting facts of interest. Which facts are "of interest" depends on the subsequent analyses. Table 2.1 shows a list of entities and relationships that are "of interest" in our research.

Entity	**Description**
cClass	Class entity
cVariable	Variable entity
cFunction	Function entity

(a) Entities of interests

Relationship	**Description**
<function1> call <function2>	<function1> calls <function2>
<function> write <variable>	assignment to <variable> appears in <function>
<function> read <variable>	value of <variable> is read in an expression in <function>
<entity1> contain <entity2>	<entity1> contains <entity2> in some form; for instance, a class contains a function
<variable1> varWrite <variable2>	<variable1> is used in an assignment to <variable2>
<variable1> parWrite <variable2>	<variable1> is used in an expression that is passed to parameter <variable2>
<variable> varInfFunc <function>	<variable> is used in a condition struture that decides whether or not <function> is called

(b) Relationships of interests

Table 2.1: Entities and relationships of interests

Variability-aware $\widehat{Rex}$ [59] is capable of parsing a C/C++ annotative SPL code base that employs feature variables and extracting program facts annotated with their presence conditions. In order to facilitate variability-aware extraction with $\widehat{Rex}$, users are required to provide an additional argument identifying the feature variables. A fact annotated with a PC holds in the subset of SPL products represented by that PC. We denote the factbase of an SPL, where some facts are annotated with PCs, as a $\widehat{factbase}$.

[2] https://clang.llvm.org

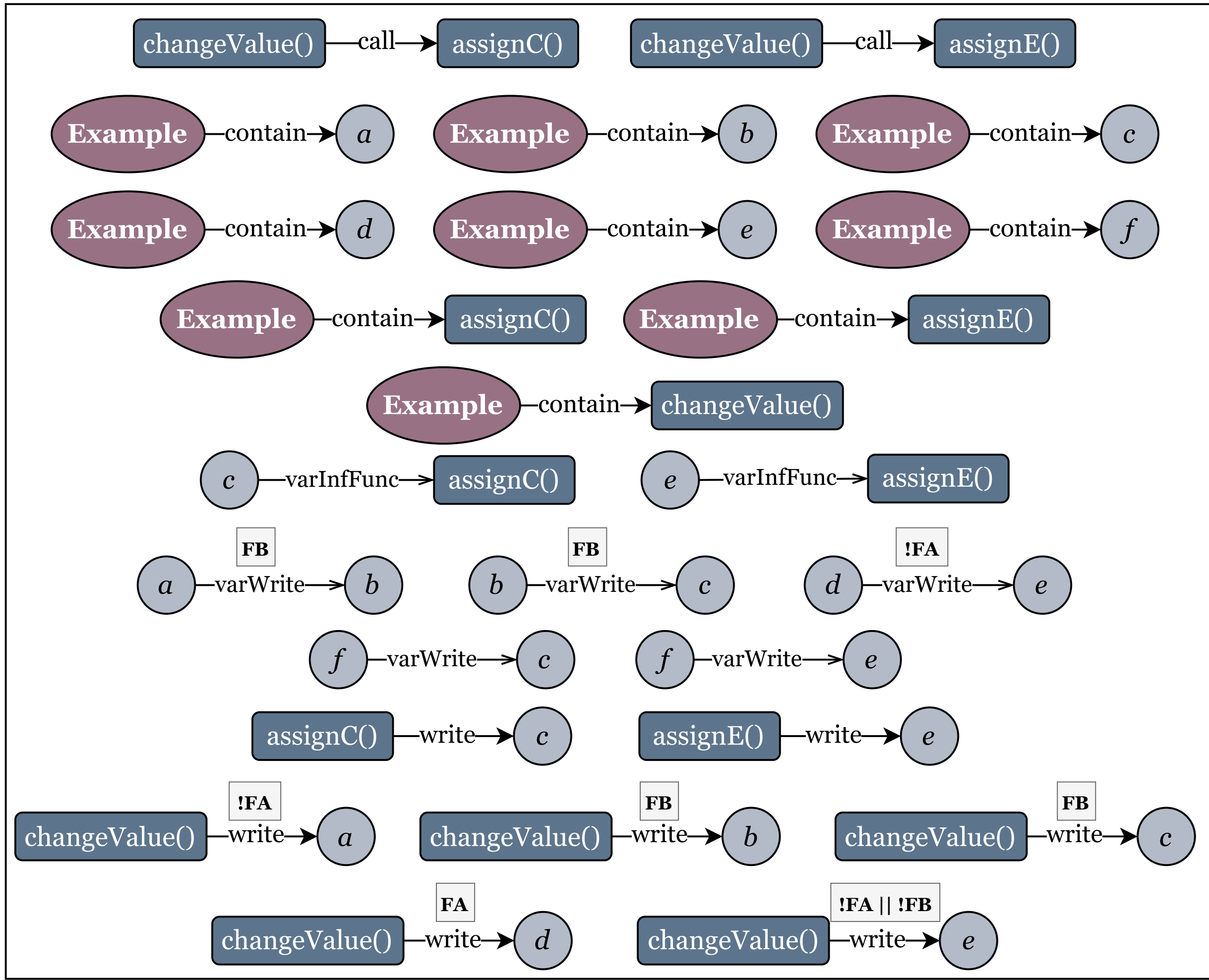


Figure 2.1: $\widehat{factbase}$ for the SPL in Listing 1.1

Example. Fig. 2.1 shows the $\widehat{factbase}$ extracted from Listing 1.1 by $\widehat{Rex}$. Variable entities appear as light-coloured circles, function entities appear as dark-coloured rounded rectangles, and class entities appear as ellipses. The relationship type *call* denotes a function call relation. The *write* relationship denotes a containment relationship - relating a *function* to a *variable* that is assigned a value in that function. The relationship type *contain* describes an entity that contains another entity in some form. For example, a class contains a function. The relationship type *varWrite* denotes a variable assignment - it relates a *variable* used in the right-hand side expression of an assignment to the *variable*

being assigned on the left-hand side of the assignment. The relationship type *varInfFunc* denotes an influence relationship - relating a *variable* to a *function* if a call to the function is conditioned on the variable. Each fact is annotated with a PC, where an empty PC denotes the *true* proposition indicating that the fact holds in every product. For example, in Fig. 2.1 none of the graph nodes have PCs because all the function and variable declarations in Listing 1.1 hold in all products.

2.3 Neo4j and Graph Database

A factbase for a software system or SPL constitutes a graphical model, where nodes represent software entities, edges represent relations between entities, and attributes record information about entities or relationships [3, 64]. Large factbases can be managed using a graph database, which is optimized for graphical data models. Many interesting software analyses can be formulated as queries on graphical software models, including reachability analyses [49], vulnerability detection [68], the management of multiple source code revisions [26], and software structure and dependency analyses [48].

```
1  <config> ::= { <relationship-filter> }? { <label-filter> }? { <min-level>
       }? { <max-level> }?
2  <relationship-filter> ::= "relationshipFilter:" ( <relationship-list> | <
      empty-string> )
3  <relationship-list> ::= <relationship> { "," <relationship> }*
4  <relationship> ::= <relationship-id> { "|" <relationship-id> }*
5  <relationship-id> ::= <string> | <string> ">" | "<" <string> | ">" | "<"
6  <label-filter> ::= "labelFilter:" ( <label-list> | <empty-string> )
7  <label-list> ::= <label> { "," <label> }*
8  <label> ::= <label-id> { "|" <label-id> }*
9  <label-id> ::= <string> | "+" <string> | "-" <string> |"/" <string> | ">"
       <string>
10 <min-level> ::= "minLevel:" <integer>
11 <max-level> ::= "maxLevel:" <integer>
12 <empty-string> ::= ""
```

Listing 2.1: BNF for config parameters of APOC path expander procedures

Neo4j [46] is a graph database that has a declarative query language *Cypher* for creating, manipulating, and querying graphs [48, 64]. Fundamentally, Cypher queries are *patterns* among graph elements. Thus users specify a graph query as a pattern of nodes, relationships and their properties. Query results can be as simple as nodes or may be

```
MATCH (srcFunc:cFunction)
CALL apoc.path.expandConfig(srcFunc, {
    relationshipFilter: 'write>',
    labelFilter: 'cVariable',
    minLevel: 1,
    maxLevel:1
})
YIELD path WITH path AS initialWrite, last(nodes(path))
    AS srcVar

CALL apoc.path.expandConfig(srcVar, {
    relationshipFilter: 'varWrite>',
    labelFilter: 'cVariable',
    minLevel: 1
})
YIELD path WITH path AS varWritePath, last(nodes(path))
    AS dstVar, initialWrite

CALL apoc.path.expandConfig(dstVar, {
    relationshipFilter: 'varInfFunc>',
    labelFilter: 'cFunction',
    minLevel: 1,
    maxLevel:1
})
YIELD path WITH path AS finalInfluence, initialWrite, varWritePath

WITH initialWrite, [varWritePath, finalInfluence] AS paths

WITH reduce(acc = initialWrite, x IN paths |
    apoc.path.combine(acc, x)) AS path

RETURN path
```

Listing 2.2: A simple query in APOC path expander procedures

paths or subgraphs. Neo4j has an extension library *APOC*[3] comprising about 450 custom procedures and functions that implement functionality not easily expressed in Cypher itself. Any query that contains both fixed- and variable-length subpatterns needs to be decomposed into subqueries (a subquery for each fixed subpattern and a subquery for each repeating subpattern). Listing 2.1 lists the Backus normal form (BNF) for configuration parameters (i.e., relationshipFilter, labelFilter, minLevel, and maxLevel) of APOC path

[3] https://neo4j.com/labs/apoc/4.3/

expander procedures that are frequently used in our examples and experimental queries. In Neo4j queries, “label” refers to the type of a node, “type” refers to the type of an edge, and “direction” refers to the direction of an edge. The *relationshipFilter* determines the relationship types and directions to traverse, the *labelFilter* determines the node labels to traverse, and *minLevel* and *maxLevel* determines the minimum and maximum number of hops in the traversal. In *relationshipFilter*, the type is represented by the string itself, and the direction is represented by “<” (incoming) or “>” (outgoing). In *labelFilter*, the label is represented by the string itself, and the filter is represented by “-” (blacklist filter), “+” (whitelist filter), “/” (termination filter), or “>” (end node filter). Syntax and examples of these four parameters can be found in the APOC documentation[4].

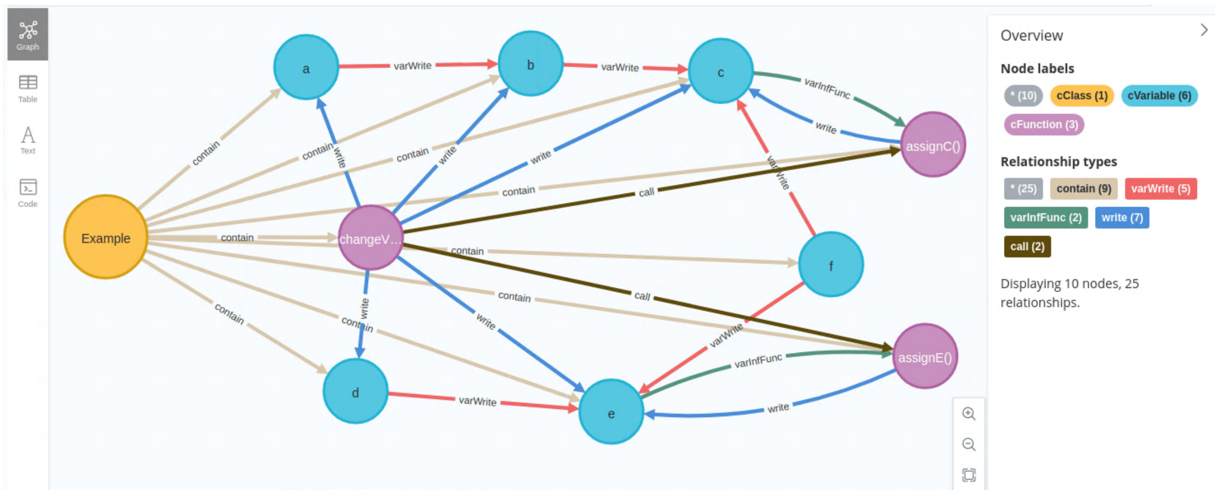


Figure 2.2: Neo4j Browser interface

Example. Listing 2.2 shows a query in APOC. This example query is a simplified version of the behaviour alternation pattern (i.e., a data-flow component interaction pattern) that is discussed in Chapter 4. The query is decomposed into three subqueries where the first and third subqueries define fixed subpatterns, whereas the second subquery specifies a repeating subpattern of arbitrary length. Each subquery expands the path result obtained from the preceding subquery, based on the provided query parameters. The first subquery (lines 2 to 9 in Listing 2.2) finds instances of the query’s initiating condition: any node of label *function* whose body writes to a node of label *variable*. The second subquery (lines 11 to 17) finds dataflows from this variable assignment to subsequent assignments, identifying variables whose values are impacted by the initial assignment. The third subquery

[4]https://neo4j.com/labs/apoc/4.3/graph-querying/expand-paths-config/

(lines 19 to 25) matches the query's terminating condition: a function call conditioned on one of the impacted variables. The APOC function *apoc.path.expandConfig*[5] (lines 2, 11, 19) expresses a path query. The full APOC query in Listing 2.2 matches the following Cypher pattern:

*(:cFunction)-[:write]->(:cVariable)-[:varWrite*1..]->(:cVariable)-[:varInfFunc]->(:cFunction)*

where *:varWrite*1..* describes a relationship pattern of relationship type *varWrite* of length 1 or more. The Cypher query looks simpler, but the APOC query is more expressive, supporting more constraints such as *whitelistNodes* and *blacklistNodes*, which specify lists of nodes to be included or excluded, respectively, in the final result.

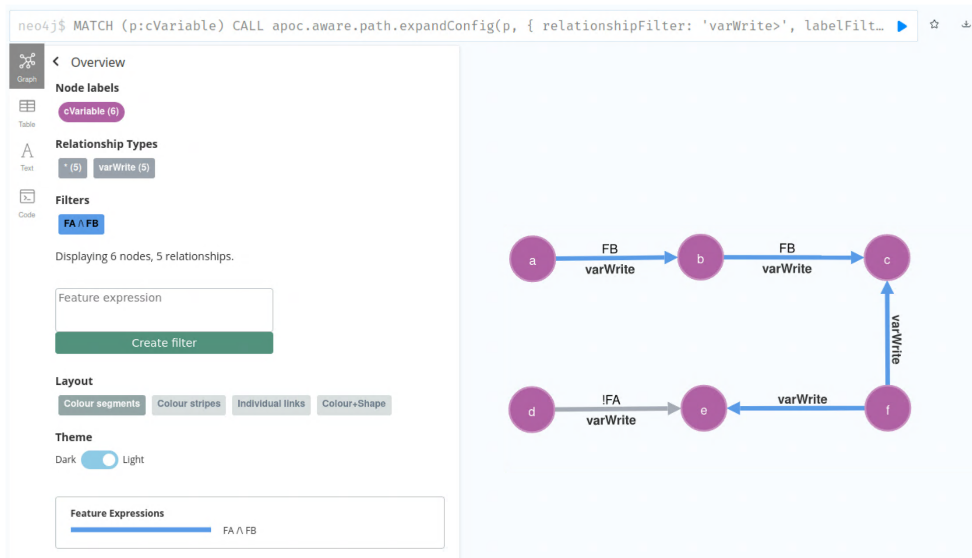


Figure 2.3: Enhancements to visualize variability-aware analysis results of $\widehat{varAssign}$ on Listing 1.1

Neo4j offers a built-in graph visualization tool called *Neo4j Browser* that allows users to visualize their data in a graphical format, facilitating the understanding of the relationships between different entities. The visualization tool in Neo4j Browser represents nodes as circles and relationships as lines or arrows connecting the nodes. Users can personalize the visual representation of nodes and relationships according to their preferences. By clicking on the nodes and relationships in the visualization, users can explore the database

[5] https://neo4j.com/labs/apoc/4.3/graph-querying/expand-paths-config/

and view the details of the selected entity. Fig. 2.2 depicts the Neo4j Browser interface visualizing the factbase in Fig. 2.1.

Toledo, one of the authors of [59], extended the Neo4j visualizer to allow the user to apply coloured filters to variability-aware results, enabling the highlighting of facts and results from the same product or set of products. Fig. 2.3 shows an example of this enhancement to visualize variability-aware analysis results of $\widehat{varAssign}$ on Listing 1.1. When applying the filter $FA \wedge FB$, the paths that are coloured blue are the subset of the $\widehat{varAssign}$ analysis results that hold in configuration $FA \wedge FB$.

2.4 Variability-aware *V-Soufflé*

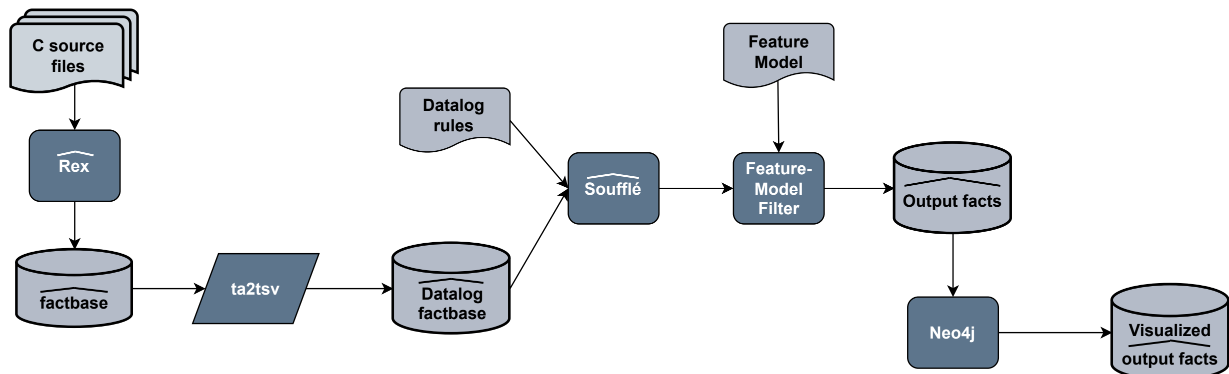


Figure 2.4: Analysis pipeline using *V-Soufflé* [59]

Shahin et al. [59] assembled a comprehensive toolchain (shown in Fig. 2.4) that (1) extracts an SPL model from an SPL's code base written in a C/C++, (2) analyzes the model, and (3) visualizes the results. First, variability-aware $\widehat{Rex}$ is applied to source code of an SPL for fact extraction, generating a $\widehat{factbase}$. Then using a Python script *ta2tsv*, the $\widehat{factbase}$ output by $\widehat{Rex}$ is converted into a Datalog $\widehat{factbase}$, which is then imported to the *V-Soufflé*[6] Datalog engine [57] along with a set of Datalog rules (i.e., analyses of interest) to perform variability-aware analysis. The analysis results from *V-Soufflé* can be optionally filtered using a *Feature Model* [33] (i.e., a representation of the set of valid feature combinations) to exclude any analysis results that do not belong to any of the valid

[6] *V-Soufflé* is an earlier name of $\widehat{Soufflé}$ [59]. In this thesis we use *V-Soufflé* to refer to the same tool.

```
.symbol_type id

// relationships
.decl cVariable (var: id)
.input cVariable

.decl cFunction (func: id)
.input cFunction

.decl write (src: id, dst: id)
.input write

.decl varWrite (src: id, dst: id)
.input varWrite

.decl varInfFunc (src: id, dst: id)
.input varInfFunc

// rules
.decl varWrite_closure(v1: id, v2: id)
varWrite_closure(v1, v2) :- varWrite(v1, v2), cVariable(v1),
                            cVariable(v2).
varWrite_closure(v1, vn) :- varWrite(v1, v2), varWrite_closure(v2, vn),
                            cVariable(v1), cVariable(v2), cVariable(vn).

.decl behav_alt(src: id, dst: id)
behav_alt(srcFunc , dstFunc) :- write (srcFunc, srcVar),
                                varWrite_closure(srcVar, dstVar),
                                varInfFunc(dstVar, dstFunc),
                                cFunction(srcFunc), cVariable(srcVar),
                                cVariable(dstVar),cFunction(dstFunc).
.output behav_alt
```

Listing 2.3: Equivalent Datalog program in Listing 2.2

SPL products. The filtered results are converted into the input format accepted by the Neo4j database for visualization.

Example. Listing 2.3 presents the equivalent Datalog program to the APOC query shown in Listing 2.2. It begins by defining the type of the *id* symbol utilized throughout the program. The Datalog rules employ the keywords *.decl* and *.input* to define a set of input relations in lines 4 to 17. The transitive closure rule for *varWrite* is defined on lines 20 to 24. Finally, lines 26 to 31 define the same example query as the Cypher/APOC query shown

```
1 "path"
2 "[{""id"":9,""labels"":[""cFunction""],""properties"":{""condition"":""""
    ,""filename"":""Example.c"",""id"":""changeValue()""}},{""id"":19,""
    type"":""write"",""start"":9,""end"":2,""properties"":{""condition"":
    ""FB"",""__csv_type"":""write""}},{""id"":2,""labels"":[""cVariable""
    ],""properties"":{""condition"":"""",""filename"":""Example.c"",""id""
    :""b""}},{""id"":14,""type"":""varWrite"",""start"":2,""end"":3,""
    properties"":{""condition"":""FB"",""__csv_type"":""varWrite""}},{""
    id"":3,""labels"":[""cVariable""],""properties"":{""condition"":"""",
    ""filename"":""Example.c"",""id"":""c""}},{""id"":11,""type"":""
    varInfFunc"",""start"":3,""end"":7,""properties"":{""condition"":""""
    ,""__csv_type"":""varInfFunc""}},{""id"":7,""labels"":[""cFunction""
    ],""properties"":{""condition"":"""",""filename"":""Example.c"",""id""
    :""assignC()""}}]"
```

Listing 2.4: Results of running the query in Listing 2.2 on the maximal product of the SPL in Listing 1.1

```
1 changeValue() assignC() @ FB
```

Listing 2.5: Results of running the query in Listing 2.3 on the maximal product of the SPL in Listing 1.1

in Listing 2.2. Line 32 outputs the query results using the *.output* keyword. Listings 2.4 and 2.5 are the results of running the APOC query in Listing 2.2 and the Datalog program in Listing 2.3 on the maximal product of the SPL in Listing 1.1, respectively.

By lifting the Neo4j reasoning engine to enable queries over an SPL $\widehat{factbase}$, we streamline the toolchain in Fig. 2.4. Expected benefits are 1) a smaller toolchain, which reduces dependencies on third-party tools like the Datalog engine, 2) richer analysis results (because Neo4j can report and visualize path results rather than just the end points of paths), and 3) better performance.

2.5 Summary

In this chapter, we introduce several concepts that are related to this thesis, including software product lines (SPLs) and annotative approaches to SPL engineering. Additionally, we present software factbases and $\widehat{factbase}$s, and we describe fact extractors, especially variability-aware $\widehat{Rex}$, which is used in our toolchain for fact extraction. We also offer an

overview of Neo4j and graph databases, with Neo4j serving as the reasoning engine for this thesis. Lastly, we introduce variability-aware *V-Soufflé*, which is the closest work related to our work.

Chapter 3

Variability-Aware $\widehat{\textbf{Neo4j}}$

In this chapter, we detail our approach to lifting the Neo4j query engine to operate on an SPL $\widehat{factbase}$ model and return variability-aware query results.

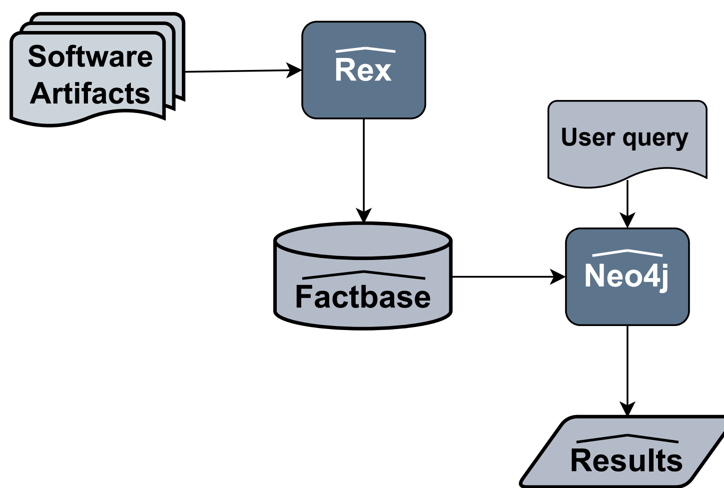


Figure 3.1: Structure of Our Toolchain

The overall toolchain for analyzing an SPL model is shown in Fig. 3.1, An SPL's source code artifacts are input to $\widehat{Rex}$ for fact extraction, producing a $\widehat{factbase}$ model of the SPL (in the format of a CSV file). $\widehat{Neo4j}$ accepts a $\widehat{factbase}$ along with a user query and generates a set of results: results can be nodes, edges, attributes (text), paths, or subgraphs in the graphical software model, each annotated with a PC representing the set of product variants for which the result applies.

3.1 Traversal Framework

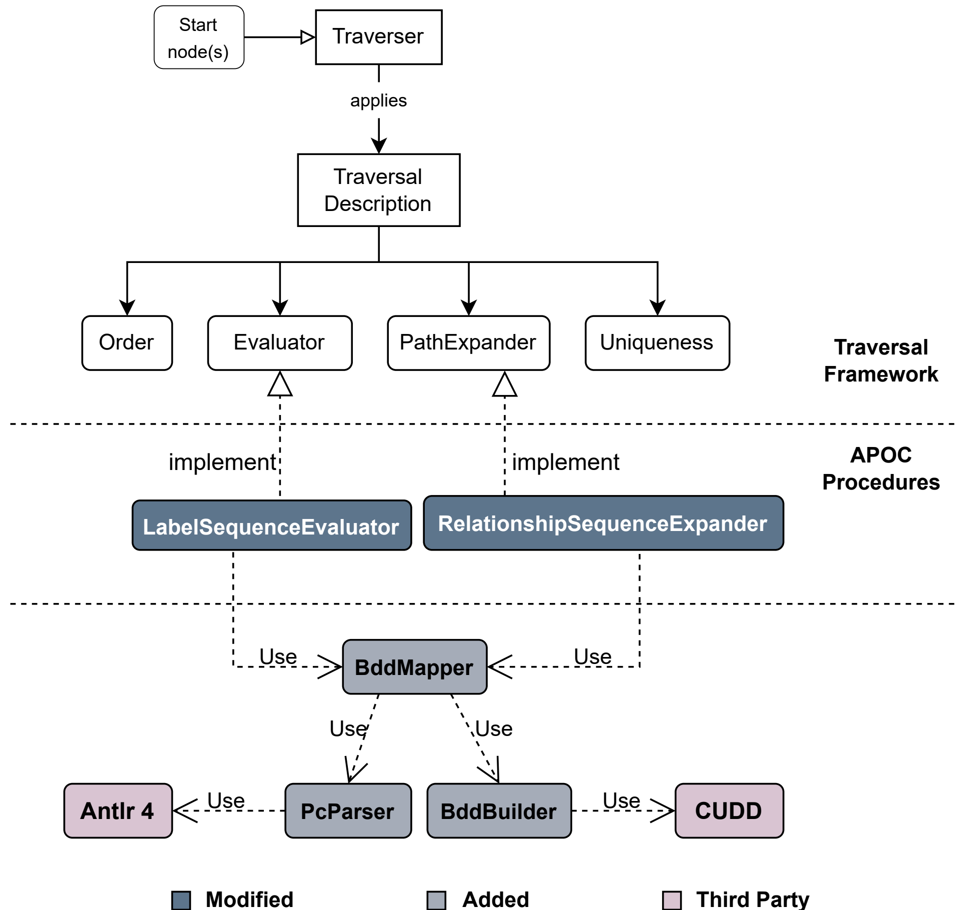


Figure 3.2: $\widehat{Neo4j}$: Modifications and Additions to Neo4j

The lifting of the Neo4j query engine focuses on the *traversal framework*[1], which traverses the graph data model and incrementally constructs the results to a graph query. The key classes of the traversal framework are depicted in the top part of Fig. 3.2. Specifically, starting from a set of start nodes, the traversal framework repeatedly expands some partially traversed path that begins at one of the start nodes and that matches the current state of the query, until the partially traversed paths cannot be further expanded.

[1] https://neo4j.com/docs/java-reference/current/traversal-framework/

A graph traversal is specified by rules encoded in the *TraversalDescription* class and its component classes *Order*, *Evaluator*, *PathExpander*, and *Uniqueness*.

- The *Order* class decides the basic traversal algorithm (i.e., breadth-first search versus depth-first search) and the branching strategy (i.e., pre-order traversal versus post-order traversal).
- The *Evaluator* class determines whether the traversal should continue from the current position (i.e., the end of the current path being traversed), and whether the current path should be included in the final result.
- The *PathExpander* class selects which edges incident on the current node should be visited, primarily based on each edge's type and direction.
- The *Uniqueness* class defines the rules for revisiting nodes and edges. By default, a node is traversed at most once.
- The *Traverser* class represents the current traversal of the graph data model in response to a query.

3.2 Lifted $\widehat{Factbase}$

Lifted $\widehat{Neo4j}$ needs to be able to query and reason over a lifted $\widehat{factbase}$, whose facts are conditionally *true* (i.e., are *true* in some product variants and *false* in others). Each fact in a $\widehat{factbase}$ is annotated with a *presence condition (PC)*, which indicates the collection of products in which the fact holds.

We use Ordered Binary Decision Diagrams (OBDDs) [14] to represent, manipulate, and evaluate PCs. Every Boolean expression has a unique *canonical* OBDD representation, which means that satisfiability checking of an OBDD is a constant-time operation (because the canonical OBDD for an unsatisfied expression is the trivial OBDD for *false*) [14]. PC annotations on facts in a $\widehat{factbase}$ are input to $\widehat{Neo4j}$ as string representations of boolean expressions. We use the *Antlr4* tool[2] to parse the string expressions and build the corresponding OBDDs. Each node and edge in $\widehat{Neo4j}$ has a PC attribute pointing to the OBDD of its PC. We use the *CUDD* OBDD library [2] to create and cache OBDDs as they are computed. The same cache stores the OBDD PCs that are computed and associated with each partial and final query result.

[2] https://www.antlr.org/

3.3 Lifted Traversal Algorithm

Variability-aware queries over a $\widehat{factbase}$ generate query results that are conditionally *true* and that are annotated with PCs. If a variability-aware query result is a set of paths, the PC of each path result is the conjunction of the PCs annotating the path's constituent nodes and edges. If a result's PC is unsatisfiable, then the result does not apply to any of the SPL's products; such results are not reported.

A brute-force method to lift Neo4j queries to be variability-aware would be to execute Neo4j's native traversal algorithm and then filter out those results whose PCs are unsatisfiable. However, this approach is inefficient because the native traversal algorithm would extend partial results even when their PCs are unsatisfiable. Instead, we augment the traversal algorithm to test the satisfiability of a partial result's PC on-the-fly as each new node and edge is added to a result path.

The result is our *lifted* traversal algorithm, which is a modified version of Neo4j's native traversal algorithm, shown in Algorithm 1. Our modifications marked in red mainly focus on caching and computing the OBDD values of the PCs. Each time the algorithm tries to expand a path result, we take the PC into account and continue to process only those paths that have a satisfiable PC, and safely remove those intermediate paths with unsatisfiable PCs because they do not apply to any of the SPL's products. It starts from a collection of *start paths* in the SPL model (rather than start nodes) and returns extended paths in the model, where the path extensions match the query pattern. The algorithm maintains a *pathWaitList[]* that stores the collection of partial results (paths) to be further expanded and a *finalResult[]* list of final query results. The query's pattern is expressed as sequences of node labels and relationship types and directions that are stored in *queryLb* and *queryRel*, respectively. Each extension's start node (i.e., the end node of the current result path) must match the query's next node label and the extension's type and direction must match the query's next relationship, indicated by the label index *li* and relationship index *ri*, respectively. If the query expresses a repeated pattern, then lines 16-17 use the modulo operator to update the indices *li* and *ri* so that they circle through the pattern[3]. The *curPath.PC* is the conjunction of the PCs of the nodes and edges along the current path. When Algorithm 1 considers a possible extension to the current path (line 23), it checks the extension's PC and it keeps the extension only if its PC is satisfiable. Thus, Algorithm 1 guarantees that the PC of each partial path added to *pathWaitList[]* is satisfiable.

[3]Index *ri* is always one step ahead of index *li* (lines 16-17) because no query-label-matching test is performed on the start node of *curPath* (line 18).

Algorithm 1: Lifted Traversal Algorithm

```
   Input  : a fact base, a set of start paths, a user query
   Output: a set of paths that match the query
 1 startPaths[] = the set of start paths
 2 pathWaitList[] = empty set of paths
 3 finalResult[] = empty set of paths
 4 queryLb[] = the sequence of label filters in the query in order
 5 queryRel[] = the sequence of relationship filters in the query in order
 6
 7 while !startPaths.isEmpty do
 8     choose Path in startPaths and remove it
 9     prefix = Path
10     curPath = prefix.endNode
11     curPath.PC = prefix.PC
12     visitedNodes[] = empty set of nodes
13     while curPath != null do
14         endNode = curPath.endNode
15         visitedNodes.add(endNode)
16         li = (curPath.length - 1) % queryLb.length //label index
17         ri = curPath.length % queryRel.length //rel index
18         if  curPath.length = 0 ∨ match(endNode.label, queryLb[li]) then
19             if curPath.length() < query.maxLevel then
20                 reList[] = endNode.relationships()
21                 matchR[] = empty set of relationships
22                 for j ← 0 to reList.length() - 1 do
23                     if !visitedNodes.contain(reList[j].endNode) ∧ matchType(reList[j].type,
                        queryRel[ri].type) ∧ matchDir(reList[j].dir, queryRel[ri].dir) ∧
                        sat(curPath.PC ∧ reList[j].PC ∧ reList[j].endNode.PC) then
24                         matchR.add(reList[j])
25                     end
26                 end
27                 if !matchR.isEmpty then
28                     for j ← 0 to matchR.length() - 1 do
29                         extension = curPath.extend(matchR[j], matchR[j].endNode)
30                         extension.PC = curPath.PC ∧ matchR[j].PC ∧ matchR[j].endNode.PC
31                         pathWaitList.add(extension)
32                     end
33                 end
34             end
35             if curPath.length() >= query.minLevel then
36                 finalResult.add(prefix.extend(curPath))
37             end
38         end
39         //BFS, DFS may choose different Path
40         choose Path in pathWaitList and remove it
41         curPath = Path // null if pathWaitList is empty
42         curPath.PC = prefix.PC ∧ Path.PC // null if curPath is null
43     end
44 end
45 return finalResult
```

During each iteration of the outer while loop, a start path is selected from *startPaths[]* and the inner while loop examines extensions to the end node of that path. Line 18 compares the label of the current path's end node to the next label in the query, and if the two labels do not match then the algorithm skips lines 18 to 38, chooses a new path from *pathWaitList[]*, and starts a new inner-loop iteration. If the labels do match, line 20 populates *reList[]* with all the relationships that are connected to the current path's end node. Lines 22 to 26 iterate through *reList[]* and add to *matchR[]* those relationships whose destination node has not yet been visited, whose type and direction match the query's next relationship and whose PC passes the satisfiability check. Lines 27 to 33 update *pathWaitList[]* with paths that extend the current path with each relationship in *matchR[]* and its destination node, and the extended paths' PCs (i.e., the length of the current path is extended by 1). Line 19 ensures that the lengths of all paths added to *pathWaitList[]* are at most *query.maxLevel*, where *maxLevel* is a constraint in the query indicating the maximum length of paths in the final result. Lines 35 to 37 ensure that any current path whose length is at least *query.minLevel* is added to *finalResult[]*.

Fig. 3.2 illustrates the key modifications and additions we made to Neo4j to implement variability-aware $\widehat{Neo4j}$: (1) We mainly modified two APOC classes *LabelSequenceEvaluator* and *RelationshipSequenceExpander*, which implement the traversal framework's *Evaluator* and *PathExpander* interfaces, respectively. (2) To implement PCs, we added class *PcParser*, which parses the input PC string expressions, and added class *BddBuilder*, which uses the CUDD library to construct and cache the corresponding OBDD PCs. After *BddMapper* maps each graph node and relationship to its OBDD, the classes *LabelSequenceEvaluator* and *RelationshipSequenceExpander* use the information in *BddMapper* to execute satisfiability checking on each partial result of the traversal algorithm, filtering out results that are unsatisfiable.

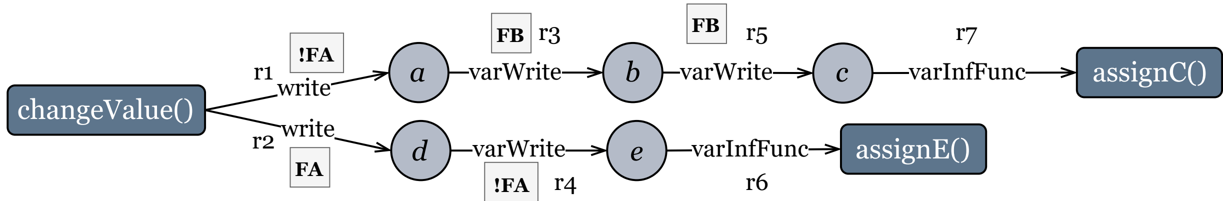


Figure 3.3: A traversal example

Example. Consider the query shown in Listing 3.1. As mentioned before, this query includes both fixed- and variable-length subpatterns, which need to be posed as distinct subqueries. The first subquery finds instances of the query's initiating condition: any

```
MATCH (srcFunction:function)
CALL apoc.aware.path.expandConfig(srcFunction, {
    relationshipFilter: 'write>',
    labelFilter: 'variable',
    minLevel: 1,
    maxLevel:1
})
YIELD path WITH path AS initialWrite

CALL apoc.aware.path.extend(initialWrite, {
    relationshipFilter: 'varWrite>',
    labelFilter: 'variable',
    minLevel: 1
})
YIELD path WITH path AS varWritePath

CALL apoc.aware.path.extend(varWritePath, {
    relationshipFilter: 'varInfFunc>',
    labelFilter: 'function',
    minLevel: 1,
    maxLevel:1
})
YIELD path RETURN path
```

Listing 3.1: Query of the traversal example

function whose body writes to a variable. The second subquery finds dataflows from this variable assignment to subsequent assignments, identifying variables whose values are impacted by the initial assignment. The third subquery matches the query's terminating condition: a function call conditioned on one of the impacted variables. Each subquery invokes a separate call of the traversal algorithm and passes to the *startPaths* parameter the partial results from the previous subquery. Table 3.1 shows the computation of the lifted traversal algorithm performing the second subquery from Listing 3.1 on the $\widehat{factbase}$, and shows the values of key variables as processing of the second subquery progresses. Fig. 3.3 shows the affected paths in the SPL model.

Initially, *startPaths[]* holds two paths computed by the first subquery: *(changeValue())-[:write]->(a)* and *(changeValue())-[:write]->(d)*. Suppose the algorithm in line 8 chooses to extend path *(changeValue())-[:write]->(a)*, assigning it to *prefix*; then *curPath* and *endNode* are both initialized to variable node *a* and *curPath.PC* is *!FA*. Line 18 evaluates to *true* because *curPath.length* is 0 (the short-circuit evaluation skips the label check). In

Table 3.1: Iterations of Algorithm 1, performing the second subquery from Listing 3.1 on the $\widehat{factbase}$ from Fig. 2.1

First iteration
startPaths[]: {~~*(changeValue())-[:write]->(a)*~~, *(changeValue())-[:write]->(d)*} **prefix:** *(changeValue())-[:write]->(a)* **curPath:** *a* **curPath.PC:** *!FA* **endNode:** *a* **reList[]:** {*r3*} **matchR[]:** {*r3*} **pathWaitList[]:** {~~*(a)-[:varWrite]->(b)*~~} **finalResult[]:** {} **choose path:** *(a)-[:varWrite]->(b)*
Second iteration
startPaths[]: {*(changeValue())-[:write]->(d)*} **prefix:** *(changeValue())-[:write]->(a)* **curPath:** *(a)-[:varWrite]->(b)* **curPath.PC:** *!FA && FB* **endNode:** *b* **reList[]:** {*r5*} **matchR[]:** {*r5*} **pathWaitList[]:** {~~*(a)-[:varWrite]->(b)-[:varWrite]->(c)*~~} **finalResult[]:** {*(changeValue())-[:write]->(a)-[:varWrite]->(b)*} **choose path:** *(a)-[:varWrite]->(b)-[:varWrite]->(c)*
Third iteration
startPaths[]: {*(changeValue())-[:write]->(d)*} **prefix:** *(changeValue())-[:write]->(a)* **curPath:** *(a)-[:varWrite]->(b)-[:varWrite]->(c)* **curPath.PC:** *!FA && FB* **endNode:** *c* **reList[]:** {*r7*} **matchR[]:** {} **pathWaitList[]:** {} **finalResult[]:** {*(changeValue())-[:write]->(a)-[:varWrite]->(b)*, *(changeValue())-[:write]->(a)-[:varWrite]->(b)-[:varWrite]->(c)*} **choose path:** null
Fourth iteration
startPaths[]: {~~*(changeValue())-[:write]->(d)*~~} **prefix:***(changeValue())-[:write]->(d)* **curPath:** *d* **curPath.PC:** *FA* **endNode:** d **reList[]:** {*r4*} **matchR[]:** {} **pathWaitList[]:** {} **finalResult[]:** {*(changeValue())-[:write]->(a)-[:varWrite]->(b)*, *(changeValue())-[:write]->(a)-[:varWrite]->(b)-[:varWrite]->(c)*} **choose path:** null

lines 22 to 26, edge *r3* is added to *matchR[]* because it matches the query's next relationship type and direction (*varWrite*>) and because $!FA \land FB$ is satisfiable. Therefore, in lines 27 to 33, *(a)-[:varWrite]->(b)* is added to *pathWaitList[]*, and *finalResult[]* remains empty because *curPath.length()* is less than *query.minLevel*. Suppose in line 40 the algorithm chooses next to extend path *(a)-[:varWrite]->(b)*; then *curPath* is set to this path and *curPath.PC* is $!FA \land FB$, and execution progresses to the next iteration.

In the second iteration of the inner while loop, the *endNode* is *b*. Line 18 evaluates to *true* because *b* matches the query's next label. In lines 22 to 26, edge *r5* is added to *matchR[]*, because it matches the query's next relationship type and direction (*varWrite*>) and because $!FA \land FB \land FB$ is satisfiable. Therefore, in lines 27 to 33, *(a)-[:varWrite]->(b)-[:varWrite]->(c)* is added to *pathWaitList[]*, and *(changeValue())-[:write]->(a)-[:varWrite]->(b)* is added to *finalResult[]*. Suppose the algorithm next chooses *(a)-[:varWrite]->(b)-[:varWrite]->(c)* to expand (line 40); then *curPath* is set to this path, *curPath.PC* is $!FA \land FB$, and execution progresses to the next iteration.

In the third iteration, the algorithm remains in the inner while loop and the *endNode* is *c*. However, none of *c*'s relationships match the type and direction of the query's next relationship (*varWrite*>), so *matchR[]* will be empty. Thus, *pathWaitList[]* will also be empty and the next *curPath* will be null. Before that happens, *(changeValue())-[:write]->(a)-[:varWrite]->(b)-[:varWrite]->(c)* is added to *finalResult[]*.

In the fourth iteration, the algorithm exits the inner while loop because *curPath* is null. Therefore, the algorithm chooses the remaining start path *(changeValue())-[:write]->(d)* as *prefix*, *curPath* and *endNode* are both initialized to the variable node *d*, and *curPath.PC* is *FA*. In subsequent lines 22 to 26, *r4* is not added to *matchR[]* because the PC of such a path extension $FA \land !FA$ is unsatisfiable, and *finalResult[]* remains unchanged. Therefore Algorithm 1 terminates exploration of this path and excludes this *curPath* from the set of *finalResult*.

3.4 Soundness

We did not assess the accuracy of the $\widehat{Neo4j}$ search results in terms of precision, recall, and F-measure because we do not know the ground truths of such analysis results for the large open-source systems used in our evaluation (i.e., whether a result is actually true, false, or conditionally true in a subset of products). Instead, we used *V-Soufflé* as an oracle to test the implementation of our $\widehat{Neo4j}$ engine because *V-Soufflé* is a well-established and respected Datalog engine.

Shahin [54] provides correctness criteria and proofs that the variability-aware reasoning in *V-Soufflé* is exactly the union of the results of the corresponding product-based reasoning applied individually to each product model. Moreover, Shahin also proves that each variability-aware result's presence condition represents exactly the set of products having this result in their un-lifted analysis results.

3.5 Summary

In this chapter, we introduce the traversal framework of Neo4j, and describe the implementation of $\widehat{Neo4j}$. We present the lifted traversal algorithm, which tests the satisfiability of a partial result's PC on-the-fly as each new node and edge is added to a result path. In addition, we introduce the modifications and additions we made to Neo4j to implement variability-aware $\widehat{Neo4j}$. In order to execute satisfiability checking on each partial result of the traversal algorithm, filtering out results that are unsatisfiable, we use the *Antlr4* tool to parse the input PC string expressions, and use the CUDD library to construct and cache the corresponding OBDD PCs.

Chapter 4

Analyses of interest

To evaluate the efficacy of $\widehat{Neo4j}$ variability-aware analyses, we identified several commonly used analyses of interest. These analyses discover potential feature interactions or conventional dataflow and control-flow patterns across multiple components in a program [45, 59, 48].

The list of analyses is presented below, and detailed information on each of them is provided in the subsequent sections.

- *Inter-component-based communication (ICBC)* [45] identifies *Direct* and *Indirect* inter-component communications between different components.
- *Loop detection (LD)* [45] is a special case of ICBC that detects whether a component communicates with itself directly or indirectly (through a cycle of communications).
- *Behaviour alternation (BA)* [45, 59] is a type of dataflow component interaction where one component assigns a new value to a variable, and this variable assignment indirectly causes another component to change its behaviour.
- *Multiple callers (MC)* [45] detects if multiple components can communicate with the same callee component.
- *Race condition (RC)* [45] detects if multiple components that communicate with the same component can lead to assignments to the same variable.
- *Direct recursion (DR)* [59] identifies functions that directly call themselves, and *Indirect recursion (IDR)* detects functions that call themselves indirectly (through a cycle of function calls).

- *Call graph analysis (CG)* [48] identifies for each function its outgoing calls and the target of those calls. The result is a graph of function calls between functions.
- *Triangle-shaped communication patterns (TSCP)* [17] identifies pattern instances in which a specific type of relationship (e.g., variable assignments or function calls) repeats three times and loops back to the starting node.

The remainder of this chapter introduces each analysis with a detailed definition. The entities, relationships, and attributes utilized in these analyses are detailed in Table 2.1.

4.1 Direct and Indirect Inter-Component-Based Communication

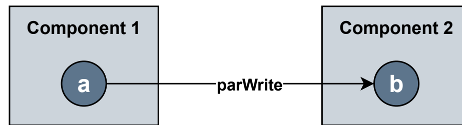


Figure 4.1: Direct inter-component-based communication pattern

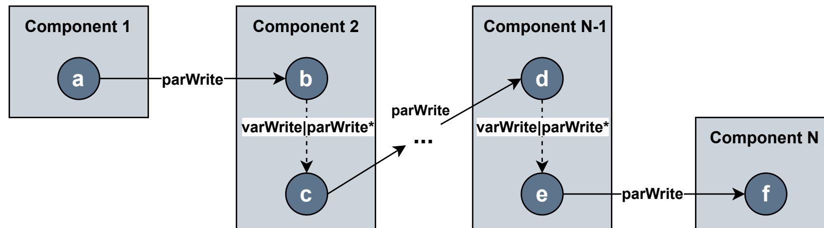


Figure 4.2: Indirect inter-component-based communication pattern

Inter-component-based communication (ICBC) is a dataflow analysis that detects a pattern of direct or indirect inter-component communications. It includes a sequence of parameter-passing function calls and/or variable assignments from the source component to the target component.

Fig. 4.1 shows the pattern of direct ICBC, which captures inter-component communications between two components, and Fig. 4.2 shows the pattern of indirect ICBC, which

captures inter-component communications between three or more components. The direct case represents a single communication via a parameter-passing function call (represented by the *parWrite* relationship) from one component to another. The indirect case refers to a dataflow scenario that involves three or more components. It begins with a parameter-passing call between components, continues within a component through variable assignments and parameter passing, and then proceeds to other components via parameter-passing function calls. Finally, it ends with an inter-component parameter-passing call.

This pattern requires that the $\widehat{factbase}$ include the following facts: (1) variable entities, (2) *parWrite* relationships, (3) *varWrite* relationships, and (4) the component information of each entity to distinguish communications between different components.

4.2 Loop Detection

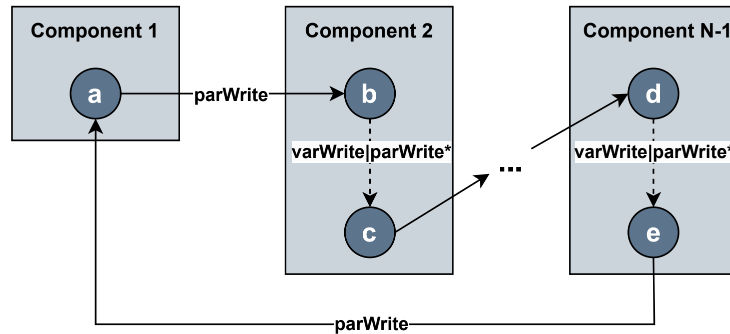


Figure 4.3: Loop detection pattern

Loop detection (LD) is a special case of inter-component-based communication in which a component communicates with itself directly or indirectly (through a cycle of communications). Fig. 4.3 shows the pattern of loop detection. It shares similarities with inter-component-based communication in terms of the communication sequence, but differs in that it involves the loopback of communication back to the source component.

This pattern requires that the $\widehat{factbase}$ include the following facts: (1) variable entities, (2) *parWrite* relationships, (3)*varWrite* relationships, and (4) the component information of each entity to distinguish communications between different components.

4.3 Behaviour Alternation

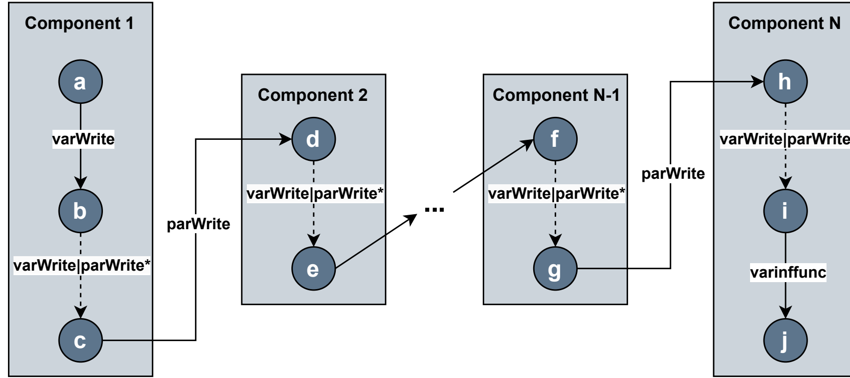


Figure 4.4: Behaviour alternation pattern

Behaviour alternation (BA) is a type of dataflow component interaction where one component assigns a new value to a variable, and this variable assignment indirectly causes another component to change its behaviour. The initial variable assignment is usually considered the *trigger* of the behaviour alternation pattern.

Fig. 4.4 shows the pattern of behaviour alternation. It starts with a variable assignment (represented as *varWrite* relationship) in a component, continues within a component through variable assignments and parameter passing, and then proceeds to other components via parameter-passing function calls. The pattern ends with an intra-component dataflow in the destination component (component N) to a control structure whose decision condition includes modified data and whose body includes a function call (represented as the *varInfFunc* relationship from the variable in the decision condition to the function being called in the control structure's body).

This pattern requires that the $\widehat{factbase}$ include the following facts: (1) variable entities, (2) function entities, (3) *parWrite* relationships, (4) *varWrite* relationships, (5) *varInfFunc* relationships, and (6) the component information of each entity to distinguish communications between different components.

4.4 Multiple Callers

Multiple callers (MC) analysis identifies scenarios in which multiple components can communicate with the same callee component. Fig. 4.5 shows the pattern of multiple callers.

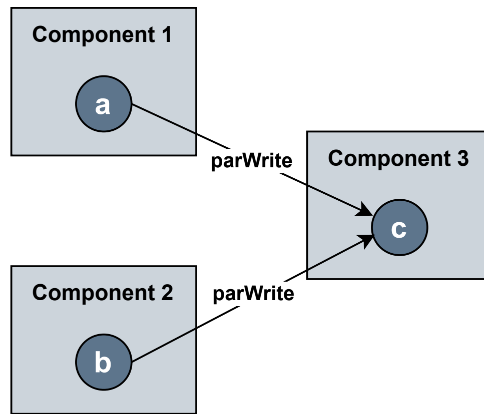


Figure 4.5: Multiple callers pattern

MC is a form of race condition in which a component (component 3 in Fig 4.5) can be called from multiple calling components simultaneously (components 1 and 2).

This pattern requires that the $\widehat{factbase}$ include the following facts: (1) variable entities, (2) *parWrite* relationships, and (3) the component information of each entity to distinguish communications between different components.

4.5 Race Condition

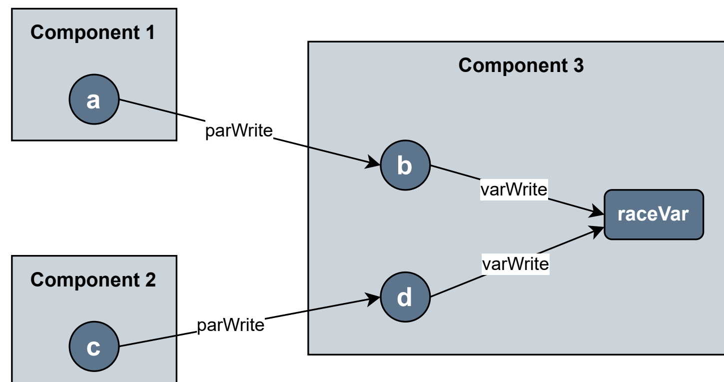


Figure 4.6: Race condition pattern

Race condition (RC) detects if multiple components that communicate with the same

component can lead to assignments to the same variable. This variable is usually called the *race variable*. Fig. 4.6 shows the pattern of a race condition. RC is an extension to the Multiple callers pattern in which two calls (from different caller components) have dataflows to the same variable (the race variable).

This pattern requires that the $\widehat{factbase}$ include the following facts: (1) variable entities, (2) *parWrite* relationships, (3) *varWrite* relationships, and (4) the component information of each entity to distinguish communications between different components.

4.6 Direct and Indirect Recursion

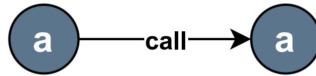


Figure 4.7: Direct recursion pattern

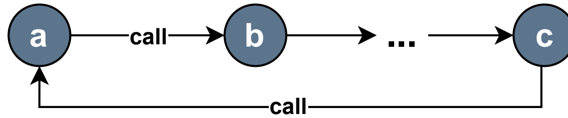


Figure 4.8: Indirect recursion pattern

Direct recursion (DR) and indirect recursion (IDR) both involve the repetitive execution of a function call. Direct recursion (please see Fig. 4.7) identifies functions that directly call themselves, and indirect recursion (please see Fig. 4.8) detects functions that call themselves indirectly (through a cycle of function calls).

Direct and indirect recursions are important subjects of study in program analysis as they can be prone to errors like stack overflows and infinite loops, which can impact program performance. Analyzing these recursion patterns provides valuable insights into the flow of program execution, enabling opportunities for improvement and optimization.

Direct and indirect recursion require that the $\widehat{factbase}$ include the following facts: (1) function entities and (2) *call* relationships.

4.7 Call Graph Analysis

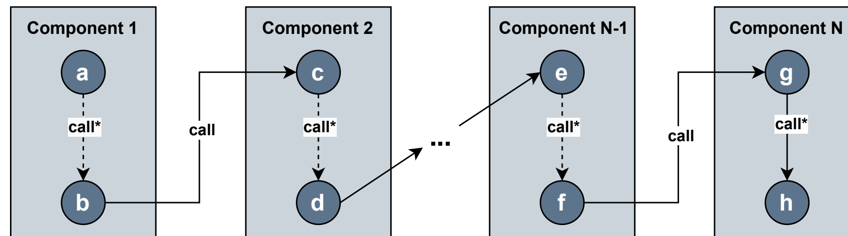


Figure 4.9: Call graph analysis pattern

Call graph analysis (CG) identifies for each function its outgoing calls and the target functions of those calls. The result is a graph of function calls between functions. By analyzing the call graph, developers and researchers can gain insights into the software system, including the program's structure. Fig. 4.9 shows the pattern of call graph analysis. This analysis requires that the $\widehat{factbase}$ include the following facts: (1) function entities and (2) *call* relationships.

4.8 Triangle-Shaped Communication Patterns

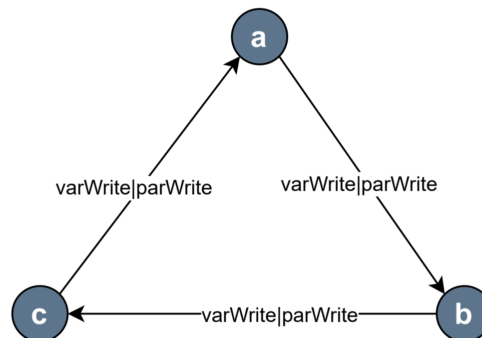


Figure 4.10: Triangle-shaped communication patterns

Triangle-shaped communication patterns (TSCP) identify pattern instances in which a specific type of relationship (e.g., variable assignments or function calls) repeats itself three

times and loops back to the starting node. Fig. 4.10 shows a triangle-shaped communication pattern of assignments (represented as *varWrite* or *parWrite* relationships among three variables).

Triangle-shaped communication analysis requires that the $\widehat{factbase}$ include the following facts: (1) variable entities, (2) *parWrite* relationships, and (3) *varWrite* relationships.

4.9 Summary

In this chapter, we provide a comprehensive summary of the ten analyses used in our evaluations. We define each analysis and outline the specific patterns it captures within the software system. Additionally, we present the necessary facts (entities, relationships, and attributes) that have to be extracted from the software and linked into the $\widehat{factbase}$ model in order to conduct these analyses.

Chapter 5

Evaluation

In this chapter, we discuss three separate evaluations that we conducted to assess our work. The first evaluation aims at comparing the performance of a post-processing approach versus an on-the-fly approach to variability-aware analysis in $\widehat{Neo4j}$. The second evaluation aims at assessing the overhead of analyzing a model of an SPL versus a model of a single product. In the third evaluation, we compare the outputs and performance of $\widehat{Neo4j}$ to a related work that employs the variability-aware *V-Soufflé* Datalog engine [59].

5.1 Experimental Setup

All three evaluations assess $\widehat{Neo4j}$ on five real-world open-source SPLs that have been used in previous variability-aware analyses projects [37, 41, 44, 50] and from an investigation conducted by Liebig et al. [40] of the variability in forty preprocessor-based SPLs. All of the subject systems use preprocessor conditional-compilation macros (`#ifdef`) to guard the feature code, whereas our model extractor $\widehat{Rex}$ expects SPL source code that uses feature variables to guard feature code. We used SugarC [47] to desugar these programs and convert the preprocessor directives into normal conditional statements (i.e., if statements).

Listing 5.1 shows an example of original source code and desugared code that results from using SugarC [1]. SugarC automatically replaces each macro-protected code block with a condition-guarded code block, where the condition is based on the macro's definition. SugarC also renames the conditions, in order to boost the readability of the desugared

[1]The renaming function and condition name in Fig. 5.1 are simplified for presentation purposes.

```
#define A 1
#define B 2

#define G(A, B)  A+B

#ifdef FA
  #define FB
#endif

#ifdef FB
  #define F(A, B) G(A, B)
#endif
...
// source code snippet
int main() {
#ifdef FB
    return F(A, B);
#endif
}
```

(a) Original source file

```
...
condition_renaming("condition_1", "!FA && FB || FA");
...
int main() {
    if (condition_1) {
        return 1 + 2;
    }
}
```

(b) Desugared file

```
...
int main() {
    if (!FA && FB || FA) {
        return 1 + 2;
    }
}
```

(c) Reverting the renaming convention

Figure 5.1: Original macro definition and modifications

code. This renaming complicates our ability to relate feature variables to conditions, to compute meaningful PCs, and to reason about PCs. Therefore, we constructed a simple script to restore the SugarC-generated condition expressions based on their `#define` macro definitions.

Three systems (axTLS [7], ToyBox [63], and BusyBox [15]) had already been desugared by the SugarC development team; thus, we included the desugared versions of these systems in our evaluation. Of the 40 potential subject systems from [40], we excluded 7 SPLs that are no longer being maintained or are mostly written in languages other than C. We excluded another 14 systems that were smaller than BusyBox (i.e., had fewer lines of code or fewer numbers of features). From the remaining 19 subject systems, we randomly selected SPLs to include in our evaluation and excluded any of these that could not be successfully desugared. In the end, we selected five nontrivial SPL subject systems for our evaluation.

Table 5.1: SPLs used in our evaluation

SPL	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
LOC(.h)	2,875	10,852	35,986	33,063	110,952
LOC(.c)	23,832	56,770	253,942	258,777	1,195,953
Components(#)	5	7	25	25	31
Features(#)	40	225	847	264	652
Facts (#)	12,972	47,849	146,441	129,009	449,189
VFacts(#)	7,308	12,025	37,027	20,442	9,252
VFacts(%)	56.34%	25.13%	25.28%	15.85%	2.06%

After desugaring the source code and restoring the SugarC condition expressions, we used $\widehat{Rex}$ to extract each SPL's $\widehat{factbase}$. Table 5.1 shows the size and factbase metrics of the subject systems. Each compilation unit represents a distinct component in a system. The count of features is determined by (1) computing the number of variables that appear in some conditional-compilation macro and (2) excluding system macros and guard macros. To get a sense of the degree of variability in the subject SPLs, we report not only the number of *Facts* extracted from each SPL but also the number of variable facts (*VFacts*) - that is, facts that are guarded by PCs.

Most analyses were executed as singleton queries. Exceptions to this were the indirect ICBC, LD, BA and CG queries, which were processed in stages to improve performance. First, we posed subqueries about potential communications within each component, and then used results about intracomponent control flows and dataflows in a full-system query.

We ran our experiments on a server with an Intel Xeon E5-2603 v4 processor (1.70GHz, 12 cores), 32GB memory, running the Ubuntu 18.04 operating system. In both evaluations, we used Neo4j Community Edition 4.3.11 and we set the heap size and page size to 30G. We ran each query five times and report the average runtime of the five executions.

5.2 Post-processing Versus On-the-fly Approach

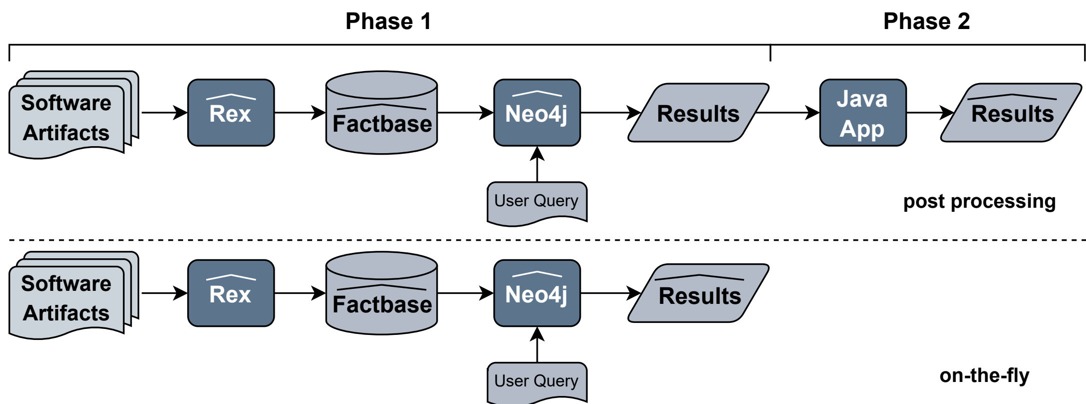


Figure 5.2: Evaluation 1

In the first evaluation, we compare the performance of two approaches to variability-aware $\widehat{Neo4j}$ queries: a post-processing approach and an on-the-fly approach (shown in Figure 5.2).

The post-processing approach uses Neo4j's native traversal algorithm as-is to find all query results in the $\widehat{factbase}$, *ignoring* the PCs on facts, and subsequently filters out the results whose PCs are unsatisfiable. The output of phase 1 will include *all* results, even the results with unsatisfiable PCs, and these results will be filtered out in phase 2. Specifically, each query result is a path of nodes and edges, each of which is annotated with a PC; and the PC of a result path is the conjunction of the PCs of all the nodes and edges in the path. If the PC of a path is not satisfiable, then the path is removed from the set of reported results.

The on-the-fly approach applies Algorithm 1 to compute each result's PC as the result is being constructed. This approach has the potential to be faster because unsatisfiable partial results are filtered out early, thereby avoiding further exploration of the path's

Table 5.2: Comparison of on-the-fly vs. post-processing of PCs on $\widehat{Neo4j}$

(a) Direct inter-component-based communication (Direct ICBC)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	462	1,466	6,378	1,151	45,888
$\widehat{Neo4j}$(ms)	31	65	188	106	786
phase 1(#)	462	1,466	6,391	1,151	45,888
phase 2(#)	462	1,466	6,378	1,151	45,888
filtered(#)	0	0	13	0	0
overhead(ms)	-92.79%	-87.03%	-78.09%	-72.11%	-60.82%

(b) Indirect inter-component-based communication (Indirect ICBC)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	518	39	405	0	37,321
$\widehat{Neo4j}$(ms)	708	52	201	122	1,895
phase 1(#)	774	39	405	0	37,321
phase 2(#)	518	39	405	0	37,321
filtered(#)	256	0	0	0	0
overhead(ms)	-27.83%	-79.61%	-58.64%	-48.31%	-25.28%

(c) Loop detection (LD)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	0	0	264	0	30,860
$\widehat{Neo4j}$(ms)	374	105	170	102	1,944
phase 1(#)	0	0	282	0	30,860
phase 2(#)	0	0	264	0	30,860
filtered(#)	0	0	18	0	0
overhead(ms)	-23.98%	-37.87%	-62.22%	-52.11%	-13.06%

(d) Behaviour alternation (BA)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	4,602	0	161	0	33,886
$\widehat{Neo4j}$(ms)	1,596	99	391	184	2,293
phase 1(#)	7,688	0	161	0	33,886
phase 2(#)	4,602	0	161	0	33,886
filtered(#)	3,086	0	0	0	0
overhead(ms)	-10.44%	-67.22%	-51.25%	-34.75%	-13.18%

(e) Multiple callers (MC)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	246	11,302	128,458	1,058	2,893,290
$\widehat{Neo4j}$(ms)	96	412	2,657	202	57,577
phase 1(#)	248	11,304	129,140	1,058	2,893,290
phase 2(#)	246	11,302	128,458	1,058	2,893,290
filtered(#)	2	2	682	0	0
overhead(ms)	-69.13%	-53.55%	-52.54%	-62.38%	-49.89%

(f) Race condition (RC)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	0	2	0	0	336
$\widehat{Neo4j}$(ms)	48	118	363	151	1,016
phase 1(#)	0	2	0	0	336
phase 2(#)	0	2	0	0	336
filtered(#)	0	0	0	0	0
overhead(ms)	-68.42%	-59.45%	-11.25%	-41.92%	-8.63%

(g) Direct recursion (DR)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	5	6	51	29	144
$\widehat{Neo4j}$(ms)	12	11	76	41	116
phase 1(#)	5	6	51	29	144
phase 2(#)	5	6	51	29	144
filtered(#)	0	0	0	0	0
overhead(ms)	-94.55%	-94.79%	-79.12%	-83.06%	-69.31%

(h) Indirect recursion (IDR)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	9	0	212	561	86
$\widehat{Neo4j}$(ms)	132	37	790	16,223	9,017
phase 1(#)	9	0	212	561	86
phase 2(#)	9	0	212	561	86
filtered(#)	0	0	0	0	0
overhead(ms)	-51.11%	-76.43%	12.86%	75.42%	71.17%

(i) Call graph analysis (CG)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	45,921	3,138	11,556	90,784	472,923
$\widehat{Neo4j}$(ms)	6,810	70	589	3,197	21,659
phase 1(#)	94,814	3,138	11,556	90,933	472,923
phase 2(#)	45,921	3,138	11,556	90,784	472,923
filtered(#)	48,893	0	0	149	0
overhead(ms)	-26.54%	-86.77%	-50.83%	-44.48%	-29.58%

(j) Triangle-shaped communication (TSCP)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	156	168	708	480	441
$\widehat{Neo4j}$(ms)	113	568	641	390	1,733
phase 1(#)	156	168	738	480	441
phase 2(#)	156	168	708	480	441
filtered(#)	0	0	30	0	0
overhead(ms)	-61.30%	-26.52%	-31.22%	-28.70%	-10.25%

possible suffixes. However, this approach could also lead to multiple satisfiability tests on an evolving PC as the PC for every partial result is tested, and these repetitive tests may be less efficient than simply testing results' PCs only at the end.

Tables 5.2a to 5.2j display the evaluation results for each query. The first section of each table shows the number of results and runtime of the on-the-fly approach employing $\widehat{Neo4j}$. The second section reports the number of results of phase 1 and phase 2. The third section reports the number of filtered results and the performance overhead of performing the on-the-fly approach (versus the post-processing approach). A negative overhead value indicates that the on-the-fly approach is more efficient. A -89.66% overhead, for

Table 5.3: Evaluation 1: Average number of results and average runtime per analysis query

	Direct ICBC	Indirect ICBC	LD	BA	MC
$\widehat{Neo4j}$(#)	55,345	38,283	31,124	38,649	3,034,354
$\widehat{Neo4j}$(ms)	235	596	539	913	12,189
filtered(#)	13	256	18	3,086	686
overhead(#)	-0.02%	-0.66%	-0.06%	-7.39%	-0.02%
overhead(ms)	-71.83%	-33.73%	-24.30%	-21.45%	-50.14%

	RC	DR	IDR	CG	TSCP
$\widehat{Neo4j}$(#)	338	235	868	624,322	1,953
$\widehat{Neo4j}$(ms)	339	51	5,240	6,465	689
filtered(#)	0	0	0	49,042	30
overhead(#)	0.00%	0.00%	0.00%	-7.28%	-1.51%
overhead(ms)	-23.74%	-81.91%	67.48%	-31.97%	-23.02%

Table 5.4: Evaluation 1: Average number of results and average runtime per program

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	51,919	16,121	148,193	94,063	3,515,175
$\widehat{Neo4j}$(ms)	992	154	607	2,072	9,804
filtered(#)	52,237	2	743	149	0
overhead(#)	-50.15%	-0.01%	-0.50%	-0.16%	0.00%
overhead(ms)	-30.14%	-62.28%	-48.58%	17.03%	-40.14%

example, indicates that the on-the-fly method is 89.66% faster than the post-processing method. Tables 5.3 and 5.4 display the aggregated results per analysis and per subject SPL, respectively.

In general, the on-the-fly approach has a faster runtime compared to the post-processing approach, and this is dependent on the number of filtered facts and the percentage of VFacts. As the number of filtered facts and percentage of VFacts increases, there is a greater likelihood that an on-the-fly analysis will be able to terminate invalid paths early, thereby improving its performance. As an example, the analysis of axTLS reveals that approximately 52,237 paths are filtered. As a result, the runtime of the on-the-fly approach is approximately 30.14% faster than the post-processing approach.

Notably, the runtime of the on-the-fly approach may be longer than the post-processing approach when the number of facts being filtered is minimal or even zero, as is the case with the indirect recursion (IDR) analysis applied to BusyBox, BerkeleyDB, and Subversion. IDR analysis returns a variable-length path, which means that the on-the-fly method is repeatedly performing satisfiability testing on the path conditions of partial results; and in the case of these programs, the partial results are never filtered out. As the path length grows, the overhead can become increasingly significant, resulting in a slower runtime for the on-the-fly approach in these situations. The path lengths of IDR results for BusyBox (range from 2 to 12), for Berkeley DB path lengths of results range from 2 to 21, and for Subversion path lengths of results range from 2 to 8.

5.3 Scalability of $\widehat{Neo4j}$

Table 5.5: $\widehat{Neo4j}$ variability-aware analysis vs. Neo4j product-based analysis

(a) Direct inter-component-based communication (Direct ICBC)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	462	1,466	6,378	1,151	45,888
$\widehat{Neo4j}$(ms)	31	65	188	106	786
product-based(#)	421	922	6,105	1,140	45,802
overhead(ms)	47.62%	38.30%	6.82%	17.78%	2.48%

(b) Indirect inter-component-based communication (Indirect ICBC)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	518	39	405	0	37,321
$\widehat{Neo4j}$(ms)	708	52	201	122	1,895
product-based(#)	330	0	348	0	37,290
overhead(ms)	161.25%	62.50%	55.81%	71.83%	22.42%

(c) Loop detection (LD)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	0	0	264	0	30,860
$\widehat{Neo4j}$(ms)	374	105	170	102	1,944
product-based(#)	0	0	234	0	30,773
overhead(ms)	144.44%	101.92%	26.87%	37.84%	37.77%

(d) Behaviour alternation (BA)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	4,602	0	161	0	33,886
$\widehat{Neo4j}$(ms)	1,596	99	391	184	2,293
product-based(#)	75	0	161	0	33,854
overhead(ms)	456.10%	39.44%	68.53%	30.50%	33.70%

(e) Multiple callers (MC)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	246	11,302	128,458	1,058	2,893,290
$\widehat{Neo4j}$(ms)	96	412	2,657	202	57,577
product-based(#)	216	5,454	120,268	1,036	2,882,290
overhead(ms)	81.13%	83.93%	12.49%	30.32%	1.88%

(f) Race condition (RC)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	0	2	0	0	336
$\widehat{Neo4j}$(ms)	48	118	363	151	1,016
product-based(#)	0	0	0	0	320
overhead(ms)	45.45%	90.32%	12.38%	12.69%	8.66%

(g) Direct recursion (DR)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	5	6	51	29	144
$\widehat{Neo4j}$(ms)	12	11	76	41	116
product-based(#)	3	5	50	29	144
overhead(ms)	71.43%	10.00%	94.87%	5.13%	18.37%

(h) Indirect recursion (IDR)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	9	0	212	561	86
$\widehat{Neo4j}$(ms)	132	37	790	16,223	9,017
product-based(#)	9	0	210	550	86
overhead(ms)	116.39%	42.31%	34.13%	63.98%	42.29%

(i) Call graph analysis (CG)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	45,921	3,138	11,556	90,784	472,923
$\widehat{Neo4j}$(ms)	6,810	70	589	3,197	21,659
product-based(#)	3,572	2,377	8,786	88,194	472,046
overhead(ms)	423.04%	25.00%	34.47%	10.97%	26.59%

(j) Triangle-shaped communication (TSCP)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	156	168	708	480	441
$\widehat{Neo4j}$(ms)	113	568	641	390	1,733
product-based(#)	156	75	597	471	414
overhead(ms)	63.77%	264.10%	6.48%	34.02%	4.27%

Our second evaluation study compares (to the extent possible) the runtimes of analyzing a model of an SPL versus separately analyzing the models of the SPL's products. Other works on SPL analysis compare the runtime of a variability-aware analysis applied to an SPL model versus the runtime of the corresponding product-based analysis applied to all of the SPL's *valid* products. However, our subject SPLs do not have feature models that specify their respective sets of valid products; and comparing against the sum of the runtimes of product-based analyses applied to *all* 2^n feature combinations in each SPL model would bias the experiment in favour of our work.

Instead, we compare $\widehat{Neo4j}$ queries against their corresponding product-based Neo4j queries on a single product: the *maximal product*, comprising all the features. The factbase for the maximal product includes all facts where feature variables are *true* and excludes all facts where any feature variable is *false*. This may result in a smaller factbase (because those facts with *false* feature variables are excluded in the factbase) and, as a result, fewer results than reported by the variability-aware query.

Table 5.6: Evaluation 2: Average number of results and average runtime per analysis query

	Direct ICBC	Indirect ICBC	LD	BA	MC
$\widehat{Neo4j}$(#)	55,345	38,283	31,124	38,649	3,034,354
$\widehat{Neo4j}$(ms)	235	596	539	913	12,189
product-based(#)	54,390	37,968	31,007	34,090	3,009,264
overhead(#)	1.76%	0.83%	0.38%	13.37%	0.83%
overhead(ms)	6.81%	45.20%	47.75%	86.55%	2.76%

	RC	DR	IDR	CG	TSCP
$\widehat{Neo4j}$(#)	338	235	868	624,322	1,953
$\widehat{Neo4j}$(ms)	339	51	5,240	6,465	689
product-based(#)	320	231	855	574,975	1,713
overhead(#)	5.63%	1.73%	1.52%	8.58%	14.01%
overhead(ms)	14.06%	32.64%	54.97%	48.37%	23.92%

Tables 5.5a to 5.5j show the experiment results for each analysis. The first section of each table shows the number of results and runtime of the variability-aware analysis employing $\widehat{Neo4j}$. The second section reports the number of results returned by the product-based Neo4j query applied to the maximal product factbase, and the runtime overhead of performing variability-aware queries. A 40.00% overhead, for example, indicates that the

Table 5.7: Evaluation 2: Average number of results and average runtime per program

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	51,919	16,121	148,193	94,063	3,515,175
$\widehat{Neo4j}$(ms)	992	154	607	2,072	9,804
product-based(#)	4,782	8,833	136,759	91,420	3,503,019
overhead(#)	985.72%	82.51%	8.36%	2.89%	0.35%
overhead(ms)	339.52%	108.83%	20.74%	50.47%	11.28%

$\widehat{Neo4j}$ query applied to the $\widehat{factbase}$ for an SPL is 40.00% slower than the product-based query. However, this is still an improvement given the number of products to be analyzed. Tables 5.6 and 5.7 display the aggregated results per analysis query and per subject SPL, respectively.

The sources of overhead include the construction of OBDDs, the computation of PCs, satisfiability testing for partial results, and the exploration of product-specific results for multiple configurations. Although satisfiability checking for OBDDs is a constant-time operation, the construction of OBDDs can have exponential time complexity in the worst case, depending on the size and complexity of the Boolean formula being represented. Each query result's PC is constructed incrementally as the result is being constructed. This approach has the potential to be faster because the analysis of infeasible results can be terminated early (as soon as the PC of a partial result becomes unsatisfiable), thereby avoiding further exploration of the partial result's possible extensions. However, this approach could also lead to multiple satisfiability tests on an evolving PC as the PC for every partial-result increment is tested.

The overhead ranges from 1.88% to 456%, in which the largest overhead appears in the behaviour alternation analysis of axTLS. Averaging over all the queries performed on all the subject $\widehat{factbase}$ models, the variability-aware analysis produces around 1.16 times more results than the maximal product analysis and has an average overhead of 106%. In general, $\widehat{factbase}$ models with greater degrees of variability (like axTLS, where more than half of the facts in the $\widehat{factbase}$ are variable facts) have higher overhead - because there are more product-specific results to compute and report.

5.4 $\widehat{\mathit{Neo4j}}$ versus *V-Soufflé*

Our third evaluation study compares the analysis results and performance of our variability-aware $\widehat{\mathit{Neo4j}}$ to variability-aware *V-Soufflé* [59] (described in Chapter 2.4). *V-Soufflé* is the most closely related work to our work, in that it analyzes a similar $\widehat{\mathit{factbase}}$ model and reports variability-aware results. However, *V-Soufflé* cannot report path results. Therefore, in order to fairly compare the tools' performances, we also ran $\widehat{\mathit{Neo4j}}$ on versions of the analyses that report endpoints of path results.

Tables 5.8a to 5.8j show the experiment results for each analysis. The first section of each table shows the number of results and runtime of the variability-aware analysis employing $\widehat{\mathit{Neo4j}}$. The second section reports the number of the end points of path results and runtime employing $\widehat{\mathit{Neo4j}}$. The last section displays the number of results and runtime of the variability-aware analysis employing *V-Soufflé*.

V-Soufflé and $\widehat{\mathit{Neo4j}}$ agree on all analyses except for three TSCP results. Specifically, *V-Soufflé* and $\widehat{\mathit{Neo4j}}$ report different PCs for these three results. To determine which analyzer is right, we used Neo4j (without the SPL analysis) to find all paths involving the nodes in the three inconsistent results and we manually computed their PCs and found them all to be unsatisfiable. Based on this, we believe that the $\widehat{\mathit{Neo4j}}$ results are correct, but it is possible that *V-Soufflé* is somehow finding other paths with different, satisfiable PCs. Definitive resolution of this inconsistency would require further investigation into how *V-Soufflé* computes paths.

With respect to performance, $\widehat{\mathit{Neo4j}}$ is usually more efficient than *V-Soufflé* when returning the same results (i.e., the end points of path results), with the exceptions of IDR and CG analyses applied to the BerkeleyDB and Subversion $\widehat{\mathit{factbase}}$. Both of these analyses explore the call paths in large models. We hypothesize that $\widehat{\mathit{Neo4j}}$ needs to find the path results in order to return the endpoints, and the performance of $\widehat{\mathit{Neo4j}}$ may suffer a combinatorial explosion when queries return long paths or when the ratio of the number of path results to the number of endpoint results is high.

Table 5.9 presents the aggregate results per analysis and Table 5.10 presents the aggregate results per subject SPL. Rows 7 to 10 in Tables 5.9 and 5.10 report the "overhead" of $\widehat{\mathit{Neo4j}}$ results compared to *V-Soufflé* results. Rows 7 and 8 of both tables present the increases in the number of results and average runtimes when $\widehat{\mathit{Neo4j}}$ reports *complete paths*. Rows 9 to 10 of both tables present the increases in the number of results and average runtimes when $\widehat{\mathit{Neo4j}}$ reports *endpoint* results. A negative runtime overhead means that

Table 5.8: Performance of $\widehat{Neo4j}$ vs. *V-Soufflé*

(a) Direct inter-component-based communication (Direct ICBC)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	462	1,466	6,378	1,151	45,888
$\widehat{Neo4j}$(ms)	31	65	188	106	786
$\widehat{Neo4j}$-endpt(#)	462	1,466	6,378	1,151	45,888
$\widehat{Neo4j}$-endpt(ms)	24	43	75	41	363
V-Soufflé(#)	462	1,466	6,378	1,151	45,888
V-Soufflé(ms)	82	170	443	145	458

(b) Indirect inter-component-based communication (Indirect ICBC)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	518	39	405	0	37,321
$\widehat{Neo4j}$(ms)	708	52	201	122	1,895
$\widehat{Neo4j}$-endpt(#)	369	23	328	0	18,493
$\widehat{Neo4j}$-endpt(ms)	241	108	260	15	475
V-Soufflé(#)	369	23	328	0	18,493
V-Soufflé(ms)	516	104	133	108	558

(c) Loop detection (LD)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	0	0	264	0	30,860
$\widehat{Neo4j}$(ms)	374	105	170	102	1,944
$\widehat{Neo4j}$-endpt(#)	0	0	157	0	14,316
$\widehat{Neo4j}$-endpt(ms)	33	17	42	21	403
V-Soufflé(#)	0	0	157	0	14,316
V-Soufflé(ms)	294	98	151	113	528

(d) Behaviour alternation (BA)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	4,602	0	161	0	33,886
$\widehat{Neo4j}$(ms)	1,596	99	391	184	2,293
$\widehat{Neo4j}$-endpt(#)	442	0	98	0	1,910
$\widehat{Neo4j}$-endpt(ms)	224	43	46	24	460
V-Soufflé(#)	442	0	98	0	1,910
V-Soufflé(ms)	383	113	310	192	653

(e) Multiple callers (MC)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	246	11,302	128,458	1,058	2,893,290
$\widehat{Neo4j}$(ms)	96	412	2,657	202	57,577
$\widehat{Neo4j}$-endpt(#)	210	10,582	126,070	1,052	2,790,036
$\widehat{Neo4j}$-endpt(ms)	26	89	742	95	12,076
V-Soufflé(#)	210	10,582	126,070	1,052	2,790,036
V-Soufflé(ms)	94	216	728	186	12,398

(f) Race condition (RC)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	0	2	0	0	336
$\widehat{Neo4j}$(ms)	48	118	363	151	1,016
$\widehat{Neo4j}$-endpt(#)	0	2	0	0	328
$\widehat{Neo4j}$-endpt(ms)	35	52	183	62	336
V-Soufflé(#)	0	2	0	0	328
V-Soufflé(ms)	92	104	328	160	661

(g) Direct recursion (DR)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	5	6	51	29	144
$\widehat{Neo4j}$(ms)	12	11	76	41	116
$\widehat{Neo4j}$-endpt(#)	5	6	51	29	144
$\widehat{Neo4j}$-endpt(ms)	7	8	20	19	26
V-Soufflé(#)	5	6	51	29	144
V-Soufflé(ms)	74	124	438	119	202

(h) Indirect recursion (IDR)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	9	0	212	561	86
$\widehat{Neo4j}$(ms)	132	37	790	16,223	9,017
$\widehat{Neo4j}$-endpt(#)	8	0	133	257	79
$\widehat{Neo4j}$-endpt(ms)	109	35	703	11,592	6,690
V-Soufflé(#)	8	0	133	257	79
V-Soufflé(ms)	128	134	714	2,672	2,150

(i) Call graph analysis (CG)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	45,921	3,138	11,556	90,784	472,923
$\widehat{Neo4j}$(ms)	6,810	70	589	3,197	21,659
$\widehat{Neo4j}$-endpt(#)	264	2,058	2,796	24,813	37,899
$\widehat{Neo4j}$-endpt(ms)	225	54	315	2,489	3,975
V-Soufflé(#)	264	2,058	2,796	24,813	37,899
V-Soufflé(ms)	397	129	547	365	731

(j) Triangle-shaped communication (TSCP)

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	156	168	708	480	441
$\widehat{Neo4j}$(ms)	113	568	641	390	1,733
$\widehat{Neo4j}$-endpt(#)	65	115	468	266	380
$\widehat{Neo4j}$-endpt(ms)	26	143	194	94	521
V-Soufflé(#)	65	115	471	266	380
V-Soufflé(ms)	94	229	628	191	586

the $\widehat{Neo4j}$ average runtime was faster. Rows 7 and 8 in Table 5.9 suggest that $\widehat{Neo4j}$ is generally slower than *V-Soufflé* when it returns complete path results, although $\widehat{Neo4j}$ can outperform *V-Soufflé* on analyses that return short fixed-length paths. Rows 9 and 10 in Table 5.9 suggest that $\widehat{Neo4j}$ is generally faster than *V-Soufflé* when it returns endpoint results, although it can be slower if the analysis explores lots of long paths. Table 5.10 suggests that in general, the runtime overhead is influenced by the size of the program's code base and the difference in the number of results returned by $\widehat{Neo4j}$ and *V-Soufflé*, respectively. For instance, the difference in the number of results from queries on ToyBox

Table 5.9: Evaluation 3: Average number of results and average runtime per analysis query

	Direct ICBC	Indirect ICBC	LD	BA	MC
$\widehat{Neo4j}$(#)	55,345	38,283	31,124	38,649	3,034,354
$\widehat{Neo4j}$(ms)	235	596	539	913	12,189
$\widehat{Neo4j}$-endpt(#)	55,345	19,213	14,473	2,450	2,927,950
$\widehat{Neo4j}$-endpt(ms)	109	220	103	159	2,606
V-Soufflé(#)	55,345	19,213	14,473	2,450	2,927,950
V-Soufflé(ms)	260	284	237	330	2,724
overhead(#)	0.00%	99.26%	115.05%	1477.51%	3.63%
overhead(ms)	-9.40%	109.87%	127.62%	176.38%	347.39%
overhead-endpt(#)	0.00%	0.00%	0.00%	0.00%	0.00%
overhead-endpt(ms)	-57.94%	-22.55%	-56.42%	-51.73%	-4.36%

	RC	DR	IDR	CG	TSCP
$\widehat{Neo4j}$(#)	338	235	868	624,322	1,953
$\widehat{Neo4j}$(ms)	339	51	5,240	6,465	689
$\widehat{Neo4j}$-endpt(#)	330	235	477	67,830	1,294
$\widehat{Neo4j}$-endpt(ms)	134	16	3,826	1,412	196
V-Soufflé(#)	330	235	477	67,830	1,297
V-Soufflé(ms)	269	191	1,160	434	346
overhead(#)	2.42%	0.00%	81.97%	820.42%	50.58%
overhead(ms)	26.10%	-73.25%	351.86%	1390.32%	99.36%
overhead-endpt(#)	0.00%	0.00%	0.00%	0.00%	-0.23%
overhead-endpt(ms)	-50.33%	-91.64%	229.92%	225.40%	-43.40%

is considerably less than the difference in the number of results from queries on axTLS, resulting in a smaller runtime overhead for $\widehat{Neo4j}$'s analysis of ToyBox.

The advantage of exposing the complete path of query results is that the engineer can examine the constituent nodes and edges of a dataflow or control-flow path result. When there are multiple path results associated with the same pair of endpoints, the engineer may gain a more comprehensive understanding of the results and their properties, and would have the ability to select the most relevant or significant path for further examination or analysis. More importantly, they can examine the PCs of the constituent nodes and edges,

Table 5.10: Evaluation 3: Average number of results and average runtime per program

	axTLS	ToyBox	BusyBox	BerkeleyDB	Subversion
$\widehat{Neo4j}$(#)	51,919	16,121	148,193	94,063	3,515,175
$\widehat{Neo4j}$(ms)	992	154	607	2,072	9,804
$\widehat{Neo4j}$-endpt(#)	1,825	14,252	136,479	27,568	2,909,473
$\widehat{Neo4j}$-endpt(ms)	95	59	258	1,445	2,533
V-Soufflé(#)	1,825	14,252	136,482	27,568	2,909,473
V-Soufflé(ms)	215	142	442	425	1,893
overhead(#)	2744.88%	13.11%	8.58%	241.20%	20.82%
overhead(ms)	360.54%	8.16%	37.24%	387.37%	418.02%
overhead-endpt(#)	0.00%	0.00%	0.00%	0.00%	0.00%
overhead-endpt(ms)	-55.90%	-58.34%	-41.63%	239.97%	33.82%

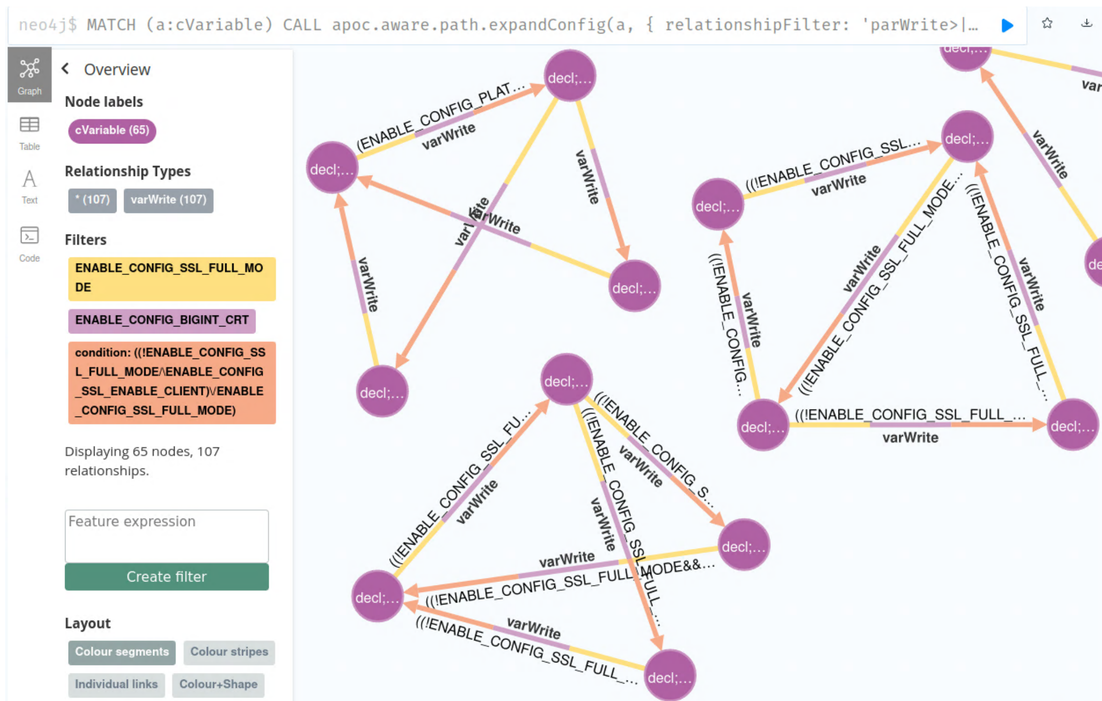


Figure 5.3: Triangle-shaped communication patterns of AxTLS

allowing the engineer to see which products contribute the entities and relationships of each path result, as shown in Fig. 5.3. This screenshot shows the extended Neo4j visualizer

[59] displaying a subset of the results of TSCP analysis applied to AxTLS. Users can use filters to highlight results that apply to specific products and can highlight the results of multiple products.

5.5 Implications and lessons learned

In general, larger $\widehat{factbase}$s lead to larger overheads for $\widehat{Neo4j}$ SPL analyses. More surprisingly, systems with similar-sized $\widehat{factbase}$s but with fewer variable facts also typically have larger overheads for SPL analyses, because when fewer facts are variable, more facts have a *true* PC, which means that these facts hold in all the products and therefore are included in more final results. It is also the case that $\widehat{factbase}$s with a larger percentage of variable facts produce a larger percentage of variable results; this is as expected.

Despite the savings achieved by lifting the Neo4j engine, variability-aware queries can still be expensive, which necessitates further performance optimizations. One possible approach would be more aggressive staging of queries (described in Chapter 4), which breaks a query down into multiple steps or subqueries. Another approach worth investigating is whether backward vs. forward traversals of the software model have a substantial impact on the runtimes of analyses.

Additionally, when developers pose queries that can potentially produce long path results (such as IDR), $\widehat{Neo4j}$ will generally have a longer runtime than *V-Soufflé* even when both analyzers report only the endpoints of path results. We hypothesize that $\widehat{Neo4j}$ needs to find the path results in order to return the endpoints. In such a case, the developer might want to use *V-Soufflé* first to report results and then use $\widehat{Neo4j}$ to investigate the paths associated with those results.

5.6 Threats to validity

There are several threats to the validity of our results. Most notably, our evaluation is limited to ten analysis queries made to five SPLs. We have tried to mitigate against this threat by employing a variety of analyses from different sources and selecting nontrivial open-source SPL systems to analyze, but it is possible that our approach may not generalize to other systems and analyses. Further evaluation of $\widehat{Neo4j}$ on larger systems and a wider range of analyses are necessary to fully understand its potential.

In our second evaluation study, our choice of using only the maximal product to approximate product performance may introduce a threat to the validity of our results. As the maximal product is likely to be the most complex and time-consuming to analyze, our evaluation of the overhead of variability-aware analyses may be biased in our favour. However, given that the measured overhead ranges from 1.88% to 456%, while the number of features in the subject SPLs ranges from 40 to 847, it is clear that the variability-aware analyses would have been found to be more efficient no matter which product was chosen for comparison. At the least, the maximal product provides a concrete baseline for comparison.

Finally, we note that the $\widehat{Rex}$ fact extractor has some limitations. It cannot extract facts about pointer addresses, function pointers, or templates due to aliasing. Moreover, because the fact extraction is based on a static analysis, certain code behaviours like threads or statement execution order may not be accurately modelled, resulting in possible false positives or negatives. However, because our evaluation studies execute $\widehat{Neo4j}$ and *V-Soufflé* on the same extracted $\widehat{factbase}$, the results of the performance evaluations remain unaffected.

5.7 Summary

In this chapter, we conducted three separate experiments to evaluate variability-aware $\widehat{Neo4j}$.

In the first experiment, we compare the performances of two approaches to variability-aware $\widehat{Neo4j}$ queries: a post-processing approach and an on-the-fly approach and show that, in general, the on-the-fly approach has a faster runtime compared to the post-processing approach.

In the second experiment, we measure (to the extent possible) the overhead of analyzing a model of an SPL versus separately analyzing the models of the SPL's products by reporting the number of results and runtime overhead of $\widehat{Neo4j}$ and the maximal product. We show that the overhead ranges from 1.88% to 456%.

In the last experiment, we compare the analysis results and performance of our variability-aware $\widehat{Neo4j}$ to variability-aware *V-Soufflé*. We found that $\widehat{Neo4j}$ is usually more efficient than *V-Soufflé* when returning the same results (i.e., the end points of path results), with the exceptions of IDR and CG analyses applied to the BerkeleyDB and Subversion $\widehat{factbase}$. We hypothesize that the performance of $\widehat{Neo4j}$ may suffer a combinatorial explosion when

queries return long paths or when the ratio of the number of path results to the number of endpoint results is high. When $\widehat{Neo4j}$ returns complete path results, it is generally slower than *V-Soufflé*, although $\widehat{Neo4j}$ can outperform *V-Soufflé* on analyses that return short fixed-length paths.

Chapter 6

Related Work

Several researchers have explored the use of graphical models of program facts for the purpose of studying questions about the modelled programs. Ebert et al. [25] employ TGraphs (i.e., directed graphs consisting of typed, attributed, and ordered nodes and edges) to represent and model source code entities (e.g., classes, functions, variables) and their relationships. They also use a graph query language called GReQL, to perform queries on the model, such as computing all caller-callee pairs. Mens et al. [42] leverage type graph and graph transformation techniques to analyze refactoring dependencies between software components. Sahu et al. [52] introduce composed control flow graphs (CCFG) to carry out dataflow testing on feature-oriented programs. However, these works have not been lifted to work on SPL models.

The growing size and complexity of modern software systems have prompted researchers to investigate the use of graph databases for storing and querying graphical models of software. Graph databases and their associated query languages provide greater flexibility and more robust querying capabilities for large graphical models compared to specific analyses. Graph databases have been used for discovering vulnerabilities [68], querying source code [64], and code comprehension [26]. Ramler et al. [48] conducted five case studies and found that although graph databases offer versatility in the data model, they do not support time series data. These works have also not been lifted to work on SPLs, but because they employ graph databases they would benefit directly from our lifted $\widehat{Neo4j}$.

Several attempts have been made to lift code-based program analyses for SPLs. Brabrand et al. [12] conduct feature-sensitive dataflow analysis on SPLs, and show that this approach is on average 5.6 times faster than the brute force approach. Bodden et al. [11] introduce SPL$^{\text{LIFT}}$ to analyze an entire SPL as a whole. However, it is a prototype restricted to Java

programs, limiting its practical application to real-world SPLs, especially those that utilize the C preprocessor. Schubert et al. [53] present VARALYZER, a novel static analysis approach that can perform inter-procedural, flow-, field-, and context-sensitive data-flow analyses on SPLs. They show that VARALYZER produces results that are consistent with the product-based approach and that it outperforms the latter in all cases. In all of these cases, the analyses scale to only small SPLs and the works lift specific analyses, whereas by lifting the Neo4j query engine we lift all analyses that can be expressed in the Neo4j query language.

The work that is most closely related to our work is the lifted *V-Soufflé* Datalog engine described in Chapter 2.4. Like our work, *V-Soufflé* queries an SPL $\widehat{factbase}$, but it uses Datalog as its query language whereas $\widehat{Neo4j}$ uses Cypher and APOC. Importantly, $\widehat{Neo4j}$ reports full path results rather than the start and end nodes of paths. Moreover, the paths' constituents are annotated by PCs, enabling the engineer to see how different products contribute to each result.

Chapter 7

Conclusion

In this thesis, we present a novel approach to analyzing graphical models of software product lines (SPLs) using a lifted $\widehat{Neo4j}$ query engine that operates on SPL $\widehat{factbases}$ and returns variability-aware query results. We conducted ten analyses on the models of five nontrivial SPLs that identify possible feature interactions, dataflows, and control-flows among program components. The overhead for obtaining analysis results for all of an SPL's products, compared to the maximum product varies from 1.88% to 456%, demonstrating the efficiency of $\widehat{Neo4j}$'s variability-aware analyses.

When compared to a competing variability-aware analyzer of SPL models, *V-Soufflé*, $\widehat{Neo4j}$ is usually more efficient when returning the same results (i.e., the endpoints of path results) and has the potential to return richer results (i.e., full path results) and to visualize variability-aware results. However, when queries that return large numbers of lengthy results are applied to large-scale systems, $\widehat{Neo4j}$ can be less efficient than *V-Soufflé*. In this case, engineers might want to use *V-Soufflé* to return endpoint results and then use $\widehat{Neo4j}$ to retrieve full path results for specific endpoints of interest.

7.1 Limitations

While our toolchain including $\widehat{Neo4j}$ demonstrates effectiveness in performing variability-aware analyses on software product lines (SPLs) and retrieving complete path results, it is essential to acknowledge that this methodology has certain limitations.

Static analysis, by its nature, involves examining code without executing it, which can lead to limitations in accurately modelling certain code behaviours. For instance, static

analysis may struggle to capture dynamic aspects such as threads or the precise order of statement execution. As a consequence, there is a possibility of generating false positives or false negatives in the analysis results.

In addition, it is important to acknowledge that the $\widehat{Rex}$ fact extractor, which is used to build the $\widehat{factbase}$s to support MDE-based analysis of SPLs, has some limitations. These limitations encompass the inability to extract facts pertaining to pointer addresses, function pointers, or templates, primarily due to challenges associated with aliasing. As a result, the fact extractor may not provide comprehensive insights into certain aspects of the analyzed code that involve pointer addresses, function pointers, or templates.

Finally, our evaluation is limited in its scope, as it is based on ten specific analysis queries conducted on five software product lines (SPLs), which necessitates caution when attempting to generalize the results. Although our three separate experiments prove the effectiveness of $\widehat{Neo4j}$ within the specific context we have focused on, it is important to acknowledge that these findings may not fully represent the diverse range of analysis scenarios and software systems.

7.2 Future Work

There are several prospective avenues for future research that build on the findings and contributions of this thesis: (1) discover more analyses of interest that can be effectively expressed in the $\widehat{Neo4j}$ query language, and apply them to larger-scale SPL systems, (2) integrate the analysis with control flow graph (CFG) blocks to increase the precision of analysis results, and (3) optimize the performance of $\widehat{Neo4j}$ analyses.

With respect to the first future extension, our evaluation is limited in that it studies only ten particular analysis queries conducted on five SPLs. We have attempted to mitigate against this limitation by selecting a variety of analyses from various sources and analyzing nontrivial open-source SPL systems. While these evaluations provide valuable insights and demonstrate the effectiveness within the specific context of our study, it is important to acknowledge that our approach may not generalize to other systems and analyses. Future research should aim to extend the evaluation of $\widehat{Neo4j}$ by identifying new analyses that can be effectively expressed using the $\widehat{Neo4j}$ query language and applying them to a larger and more diverse set of SPLs to ensure broader applicability and validity of the findings and to fully understand the potential of $\widehat{Neo4j}$.

Due to the limitations of static analysis, it is also important to consider how to reduce the number of false positive and false negative results. One potential avenue for improving

the precision of our analyses is to integrate facts about an SPL's control flow graph (CFG) block and verify that analysis results correspond to actual control flows in the code.

Lastly, as the size and complexity of SPL systems increase, it becomes essential to dive deeper into optimizing the performance of $\widehat{Neo4j}$ analyses. One possible method is to explore more systematically the effect of staging queries on performance. Another approach worth investigating is the degree to which backward versus forward traversals of the software model impact query runtimes. By analyzing the impact of these query-processing strategies, we can better understand their advantages and trade-offs in analyses. Such investigations would provide insights into how best to leverage the software model's structure and dependencies to optimize analyses. The goal would be to identify more efficient query strategies and traversal techniques and produce more precise results while minimizing computational overhead.

References

[1] Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wasowski. Variability bugs in highly configurable systems: a qualitative analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM'18)*, 26(3):1–34, 2018.

[2] ADD-Lib. The Java Library for Algebraic Decision Diagrams. Accessed: April 10, 2023.

[3] Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Computing Surveys (CSUR'08)*, 40(1):1–39, 2008.

[4] antlersoft. Browse-by-query (bbq), 2005.

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines*. Springer, 2016.

[6] Sven Apel, Alexander Von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *2013 35th International Conference on Software Engineering (ICSE'13)*, pages 482–491. IEEE, 2013.

[7] axTLS. axTLS Embedded SSL. Accessed: April 10, 2023.

[8] Shoham Ben-David, Baruch Sterin, Joanne M Atlee, and Sandy Beidu. Symbolic Model Checking of Product-Line Requirements Using SAT-Based Methods. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE'15)*, volume 1, pages 189–199. IEEE, 2015.

[9] Danilo Beuche, Michael Schulze, and Maurice Duvigneau. 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, pages 262–269, 2016.

[10] Shuvra S Bhattacharyya, Ed F Deprettere, and Bart D Theelen. Dynamic dataflow graphs. *Handbook of Signal Processing Systems*, pages 905–944, 2013.

[11] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. SPLLIFT: Statically Analyzing Software Product Lines In Minutes Instead Of Years. *ACM SIGPLAN Notices (SIGPLAN'13)*, 48(6):355–364, 06 2013.

[12] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, and Paulo Borba. Intraprocedural Dataflow Analysis for Software Product Lines. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development (AOSD'12)*, pages 13–24, 2012.

[13] Martin Bravenboer and Yannis Smaragdakis. Strictly Declarative Specification of Sophisticated Points-To Analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA '09)*, pages 243–262, 2009.

[14] Randal E Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers (TC'86)*, C-35(8):677–691, 08 1986.

[15] BusyBox. BusyBox: The Swiss Army Knife of Embedded Linux. Accessed: April 10, 2023.

[16] Gerardo Canfora, Aniello Cimitile, and Ugo De Carlini. A logic-based approach to reverse engineering tools production. *IEEE Transactions on Software Engineering*, 18(12):1053–1064, 1992.

[17] Vito Giovanni Castellana, Marco Minutoli, Shreyansh Bhatt, Khushbu Agarwal, Arthur Bleeker, John Feo, Daniel Chavarría-Miranda, and David Haglin. High-Performance Data Analytics Beyond the Relational and Graph Data Models with GEMS. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'17)*, pages 1029–1038. IEEE, 2017.

[18] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1 (ICSE'10)*, pages 335–344, 2010.

[19] Paul Clements and Linda Northrop. *Software Product Lines.* Addison-Wesley Boston, 2002.

[20] Tal Cohen, Joseph Gil, and Itay Maman. Jtl: the java tools language. *ACM SIGPLAN Notices*, 41(10):89–108, 2006.

[21] CPPX. CPPX - C++ Fact Extractor. Accessed: April 10, 2023.

[22] Ian J Davis and Michael W Godfrey. From Whence It Came: Detecting Source Code Clones by Analyzing Assembler. In *2010 17th Working Conference on Reverse Engineering (WCRE'10)*, pages 242–246. IEEE, 2010.

[23] Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. The soul tool suite for querying programs in symbiosis with eclipse. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 71–80, 2011.

[24] Jürgen Ebert, Bernt Kullbach, and Andreas Winter. Grax-an interchange format for reengineering tools. In *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*, pages 89–98. IEEE, 1999.

[25] Jürgen Ebert, Volker Riediger, and Andreas Winter. Graph Technology in Reverse Engineering The TGraph Approach. *10th Workshop Software Reengineering (WSR'08)*, P-126:67–81, 2008.

[26] Oshini Goonetilleke, David Meibusch, and Ben Barham. Graph Data Management of Evolving Dependency Graphs for Multi-Versioned Codebases. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*, pages 574–583. IEEE, 11 2017.

[27] Sergio Greco and Cristian Molinaro. Datalog and logic databases. *Synthesis Lectures on Data Management*, 7(2):1–169, 2015.

[28] Soonhoi Ha and Hyunok Oh. Decidable dataflow models for signal processing: Synchronous dataflow and its extensions. In *Handbook of Signal Processing Systems*, pages 1083–1109. Springer, 2013.

[29] Robert Hackman, Joanne M Atlee, Alistair Finn Hackett, and Michael W Godfrey. mel-Model Extractor Language for Extracting Facts from Models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS'20)*, pages 200–210, 2020.

[30] CE Hrischuk. Principles for the automated construction of distributed application software execution models.

[31] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, 2003.

[32] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification (CAV'16)*, pages 422–430. Springer, 07 2016.

[33] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[34] Christian Kästner and Sven Apel. Virtual Separation of Concerns - A Second Chance For Preprocessors. *Journal of Object Technology (JOT'09)*, 8(6):59–78, 09 2009.

[35] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type Checking Annotation-Based Product Lines. *ACM Transactions on Software Engineering and Methodology (TOSEM'12)*, 21(3):1–39, 2012.

[36] Christian Kästner, Paolo G Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA'11)*, pages 805–824, 10 2011.

[37] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A Variability-Aware Module System. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA'12)*, pages 773–792, 2012.

[38] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: A tool framework for feature-oriented software development. In *2009 IEEE 31st International Conference on Software Engineering*, pages 611–614. IEEE, 2009.

[39] Jens Knodel and Martin Pinzger. Improving Fact Extraction of Framework-Based Software Systems. In *10th Working Conference on Reverse Engineering (WCRE'03)*, pages 186–186. IEEE Computer Society, 2003.

[40] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1 (ICSE'10)*, pages 105–114, 2010.

[41] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)*, pages 643–654. IEEE, 2016.

[42] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing Refactoring Dependencies Using Graph Transformation. *Software & Systems Modeling (SoSyM'07)*, 6(3):269–285, 2007.

[43] Jan Midtgaard, Aleksandar S Dimovski, Claus Brabrand, and Andrzej Wasowski. Systematic Derivation of Correct Variability-Aware Program Analyses. *Science of Computer Programming (SCP'15)*, 105:145–170, 07 2015.

[44] Austin Mordahl. Toward Detection and Characterization of Variability Bugs in Configurable C Software: An Empirical Study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion'19)*, pages 153–155. IEEE, 2019.

[45] Bryan J Muscedere, Robert Hackman, Davood Anbarnam, Joanne M Atlee, Ian J Davis, and Michael W Godfrey. Detecting Feature-Interaction Symptoms In Automotive Software Using Lightweight Analysis. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, pages 175–185. IEEE, 02 2019.

[46] Neo4j. Neo4j Graph Data Platform. Accessed: April 10, 2023.

[47] Zachary Patterson, Zenong Zhang, Brent Pappas, Shiyi Wei, and Paul Gazzillo. SugarC: Scalable Desugaring of Real-World Preprocessor Usage into Pure C. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*, pages 2056–2067, 2022.

[48] Rudolf Ramler, Georg Buchgeher, Claus Klammer, Michael Pfeiffer, Christian Salomon, Hannes Thaller, and Lukas Linsbauer. Benefits and drawbacks of representing and analyzing source code and software engineering artifacts with graph databases. In *Software Quality: The Complexity and Challenges of Software Engineering and Software Quality in the Cloud (SWQD'2019)*, pages 125–148. Springer, 2019.

[49] Thomas Reps. Program Analysis via Graph Reachability. *Information and software technology (IST'98)*, 40(11-12):701–726, 1998.

[50] Alexander Von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Transactions on Software Engineering and Methodology (TOSEM'18)*, 27(4):1–33, 2018.

[51] Martin P Robillard and Gail C Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM'07)*, 16(1):3–es, 2007.

[52] Madhusmita Sahu and Durga Prasad Mohapatra. Data Flow Testing of Feature-Oriented Programs. *International Journal of System Assurance Engineering and Management (IJSAEM'22)*, 13(5):2291–2306, 2022.

[53] Philipp Dominik Schubert, Paul Gazzillo, Zach Patterson, Julian Braha, Fabian Schiebel, Ben Hermann, Shiyi Wei, and Eric Bodden. Static Data-Flow Analysis for Software Product Lines in C: Revoking the Preprocessor's Special Role. *Automated Software Engineering (ASE'22)*, 29(1):35, 2022.

[54] Ramy Shahin. *Language-Based Lifting of Analyses to Software Product Lines.* PhD thesis, University of Toronto (Canada), 2021.

[55] Ramy Shahin, Murad Akhundov, and Marsha Chechik. Annotative Software Product Line Analysis Using Variability-aware Datalog. *IEEE Transactions on Software Engineering (TSE'23)*, 49(3):1323–1341, 03 2023.

[56] Ramy Shahin and Marsha Chechik. Automatic and Efficient Variability-Aware Lifting of Functional Programs. *Proceedings of the ACM on Programming Languages (PACMPL'20)*, 4(OOPSLA):1–27, 11 2020.

[57] Ramy Shahin and Marsha Chechik. Variability-aware datalog. In *International Symposium on Practical Aspects of Declarative Languages*, pages 213–221. Springer, 2020.

[58] Ramy Shahin, Marsha Chechik, and Rick Salay. Lifting Datalog-Based Analyses to Software Product Lines. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*, pages 39–49, 2019.

[59] Ramy Shahin, Rafael Toledo, Robert Hackman, Joanne M Atlee, and Marsha Chechik. Applying Declarative Analysis to Industrial Automotive Software Product Line Models. *Empirical Software Engineering (EMSE'23)*, 28(2):40, 2023.

[60] M-AD Storey, Kenny Wong, Philip Fong, D Hooper, Kory Hopkins, and Hausi A Muller. On designing an experiment to evaluate a reverse engineering tool. In *Proceedings of WCRE'96: 4rd Working Conference on Reverse Engineering*, pages 31–40. IEEE, 1996.

[61] Margaret-Anne D Storey, Kenny Wong, and Hausi A Müller. Rigi: A visualization environment for reverse engineering. In *Proceedings of the 19th international conference on Software engineering*, pages 606–607, 1997.

[62] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification And Survey Of Analysis Strategies For Software Product Lines. *ACM Computing Surveys (CSUR'14)*, 47(1):1–45, 07 2014.

[63] ToyBox. Toybox: All-In-One Linux Command Line. Accessed: April 10, 2023.

[64] Raoul-Gabriel Urma and Alan Mycroft. Source-Code Queries with Graph Databases - With Application to Programming Language Usage and Evolution. *Science of Computer Programming (SCP'15)*, 97:127–134, 2015.

[65] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual Southeast regional conference*, pages 1–6, 2010.

[66] Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. The PLA Model: On the Combination of Product-Line Analyses. In *International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13)*, pages 1–8, 01 2013.

[67] David M Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[68] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy (SP'14)*, pages 590–604. IEEE, 11 2014.