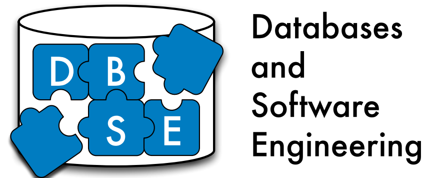
University of Magdeburg

Department of Computer Science



Master's Thesis

[Lightweight, Variability-Aware Change Impact Analysis]

Author:

[Muhammad Umar] [Ashraf]

[November 09, 2018]

Advisors:

Dr.-Ing *[Sandro Schulze]*

Faculty of computer science (FIN)

Prof. Dr. *[Gunter Saake]*

Faculty of computer science (FIN)

***[Ashraf]*, *[Muhammad Umar]*:**
[Lightweight, Variability-Aware Change Impact Analysis]
Master's Thesis, University of Magdeburg, *[2018]*.

Declaration of Authorship

I, ***Muhammad Umar Ashraf***, declare that this thesis titled, ***"Lightweight, Variability-Aware Change Impact Analysis"*** and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed: ____

Date: ____

Be kind, for whenever kindness becomes part of something, it beautifies it. Whenever it is taken from something, it leaves it tarnished.

Muhammad *P.B.U.H*

Abstract

In *Product Line Engineering (PLE)* collection of similar software systems are created using common code base and shared set of software assets to speed the development process and minimize the cost as well as time-to-market. These systems are termed as variants in *Software Product Lines (SPLs)*, each variant distinguishes from other variants on the bases of *Feature* (commonalities and distinctions) selection. These features are managed at the source code level using some variability mechanisms such as plug-ins, separate modules or annotations. By the passage of time, the number of variants of a product line increases, and therefore, it becomes impossible to test all variants every time some change has been made to the common code base or even when a new release candidate is ready. Hence, it would be nice to know which variants are affected by a set of changes. The focus of this thesis is to implement an efficient and lightweight *Change Impact Analysis (CIA)* technique which discovers all affected features, given a set of changes. Then, based on these known features determine the affected variants.

Acknowledgements

I would first like to render my warmest thanks to my main mentor and thesis supervisor *Dr.-Ing. Sandro Schulze*, who made this work possible. His friendly guidance and expert advice have been invaluable throughout all stages of the work. He consistently allowed me with full freedom so that the paper solely reflects my own work. Having said that, he also steered me in the right direction whenever he thought I lacked it.

I would also wish to extend my gratitude to *Dr. Mustafa Al-Hajjaji*, who has given his valuable time as the second reviewer for this work, and I am gratefully indebted for his worthy comments on this thesis. I also gratefully acknowledge *Prof. Dr. Gunter Saake*, for providing me this opportunity of writing this thesis under his supervision.

Finally, I must express my very profound gratitude to my parents, my siblings, other family members, and sincere friends for all their prayers, encouragement, and moral support throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Author
Muhammad Umar Ashraf

Contents

List of Figures **xiii**

List of Tables **xv**

List of Code Listings **xvii**

1 Introduction **1**
1.1 Goal of this Thesis . . . 2
1.2 Structure of the Thesis . . . 2

2 Background **5**
2.1 Characteristics of variable software systems . . . 5
2.2 Variability Implementation Mechanisms . . . 7
2.2.1 Composition Based Approach . . . 7
2.2.2 Annotation Based Approach . . . 8
2.3 Variability-aware Program Analysis . . . 10
2.4 Change Impact Analysis . . . 10
2.4.1 Dynamic CIA . . . 11
2.4.2 Static CIA . . . 11
2.4.3 Program Slicing . . . 11

3 Concept and Implementation **15**
3.1 Concept . . . 15
3.1.1 SrcML . . . 15
3.1.2 SrcSlice . . . 18
3.1.3 cppstats . . . 20
3.2 Implementation . . . 23
3.2.1 Extraction . . . 25
3.2.2 Analysis & Normalization . . . 25
3.2.3 Comparison . . . 25
3.2.4 Slicing . . . 27
3.2.5 Mapping . . . 27
3.2.5.1 Analyze affected slice . . . 27
3.2.5.2 Analyze affected feature . . . 29

4 Evaluation **33**
4.1 Research Questions . . . 33
4.2 Subject Systems . . . 34

4.2.1 OpenVPN . . . 34
4.2.2 BusyBox . . . 34
4.2.3 Vim . . . 35
4.3 Methodology . . . 36
4.3.1 Evaluation Metrics . . . 37
4.4 Results . . . 38
4.4.1 Arguing RQ’s . . . 40
4.4.1.1 *RQ1* . . . 40
4.4.1.2 *RQ2* . . . 41
4.4.1.3 *RQ3* . . . 43
4.4.2 Summing-up . . . 44

5 Related Work **45**

6 Conclusion and Future Work **47**

Bibliography **49**

List of Figures

2.1 Feature model describing a car SPL. 6

3.1 Illustration of our proposed CIA implementation. 24

4.1 *CIA* results for *OpenVPN*. 39
4.2 *CIA* results for *BusyBox*. 39
4.3 *CIA* results for *Vim*. 40
4.4 Percentage of *NIF* w.r.t *NFL*. 42
4.5 Percentage of *NCIF* w.r.t *NIF*. 43

List of Tables

3.1 `cppstats_featurelocations` table with sample data. 22
3.2 `listoffeatures` table with sample data. 22

4.1 Latest meta information of the subject systems 35
4.2 Meta data of the *OpenVPN* selected commits 36
4.3 Meta data of the *BusyBox* selected commits 37
4.4 Meta data of the *Vim* selected commits 38
4.5 Features impacted during latest commit comparisons 41
4.6 Frequently impacted features 44

List of Code Listings

2.1 A Simple C program example, illustrating different preprocessor types. 9
2.2 Conditional Inclusion 9
2.3 Result after *CPP* 9
2.4 A program fragment to be backward sliced. 12
2.5 The backward slice for variable `average` at line 17 in Listing 2.4. . . . 12
2.6 A program fragment to be forward sliced. 13
2.7 The backward slice for variable `sum` at line 12 in Listing 2.6. 13
2.8 The forward slice for variable `sum` at line 3 in Listing 2.6. 14
3.1 A C program file to be convert to srcML format. 16
3.2 A C++ program fragment after applying srcML. 17
3.3 A sample C++ program file to be SrcSlice 18
3.4 Slices produced after performing srcSlice on program in Listing 3.3 . . 19
3.5 Sample source file illustrating code to be analyzed with *Featurelocation*. 21
3.6 Old Revision 26
3.7 New Revision 26

1. Introduction

During the evolution of any software system, maintenance is considered to be the most difficult phase in software development. As stated in the laws of software evolution by Lehman and Belady, the very first law they proposed is - *change is continual* [32]. Due to a wide range of change requirements to meet customer needs, it is always challenging for developing, testing, analyzing and maintaining a software system. When a system undergoes through changes it will inevitably encounters some unexpected effects which may cause inconsistencies with other components of that particular software system, therefore, it requires a mean to thoroughly test the impact made by those changes.

Research on variable software systems has advanced significantly, and therefore, *Software Product Line (SPL)* engineering has gained substantial esteem in the industry. In SPL, a family of software products shares a common set of features. *Features* are basically used to distinguish the products of a particular product line and bridge the gap between the customer requirements and the product functionalities [3], this paradigm in SPL is also referred as *Feature-oriented software development(FOSD).* While developing a set of related software variants, developers have to ensure that they all behave as expected, however, it has been observed that even for small systems, analyzing and testing all possible variants is computationally infeasible because of the exponential number of variants. Change impact analysis (*CIA*) is a robust and efficient mean to identify the impact of source code changes, that is, to spot the potential consequences of a proposed change to a software system. *CIA* is not only helpful for developers in tracing the rippling effects after a change has been made, but is turned out to be a fine remedy in regression testing [28]. Also, it facilitates developers to better understand programme or predict impact of change and cost estimation before making a change [32][8]. Since many existing *CIA* based techniques do not take variability into account, therefore, in this thesis we proposed such a variability-aware *CIA* implementation, which helps to identify affected features, and thus, the variants containing those features.

1.1 Goal of this Thesis

Change impact analysis (*CIA*) plays a crucial role in the analysis of a program and also evidently proves to be of great relevance with product line engineering (*PLE*). It recognizes all possible implications and consequences after applying a certain change, or, estimating change impact and expense before applying a certain change [2]. The goal of this thesis is to develop a tool for determining possibly impacted variants when changing common code base of a product line, some pivotal aspects of our proposed tool are as following:

- Provides a lightweight technique to improve *CIA* support for variable software systems.

- Provides an efficient way to cope with systems in which source code is annotated directly with variability information e.g. product lines which incorporate preprocessors as a mean for annotation.

- Provides more precise results than many existing *CIA* approaches.

Our proposed tool is mainly based on a lightweight, highly scalable, robust, multi-language parsing tool called *srcML*, which provides us with an abstract program representation in XML format. Based on this representation, we extracted all the necessary information in order to compute a refined set of impacted statements, provided a particular change. We also evaluated our proposed tool against different mid and large-scale open source systems, which we discussed in the later chapters.

1.2 Structure of the Thesis

The thesis is organized into the following sections:

- Background (Chapter 2): Shed light on the context of this research, also provides a brief overview of fundamental concepts essential for better understanding of the thesis.

- Concept and Implementation (Chapter 3): Presents our variability-aware *CIA* solution, its implementation in detail with the help of a Pseudocode description, the third party tools that we have employed in the process.

- Evaluation (Chapter 4): Shows the outcome of our approach after evaluating it against different opensource variable software systems to measure the benefits of our proposed variability-aware *CIA*.

- Related Work (Chapter 5): Presents other research works that have been done in the respective area, also give brief comparison with our approach.

- Conclusion and Future Work (Chapter 6): Draw conclusion from our research findings and implications, also present our standpoint on future related works.

2. Background

In this chapter, we briefly elucidate some basic fundamental concepts that are centrally relevant and vital to better understand characteristics of variable software systems, specifically *SPL*. In particular, we highlight the concept of *Features* in software product lines and briefly look through different variability implementation mechanisms, such as plugins, frameworks, module systems, or annotations. We also shed light over the different existing variability-aware program analysis techniques and tools. Finally, last but not least, we look through the concept and characteristics of *Change impact analysis (CIA)* and briefly review some existing approaches.

2.1 Characteristics of variable software systems

The traditional software engineering processes aim solely on the development life cycle of a single software system, that is, developers gather requirements, then design and implement the target system either in consecutive, separate phases or in agile cycles [3]. However, in software product lines while analyzing and implementing a single software system, one has to glance at a variety of potential systems that could be alike yet not identical. In a nutshell, software product lines (SPL) exhibits a family of products in an application domain.

In feature-oriented software product lines paradigm, the concept of *Feature* is of foremost importance. Features are used to define the characteristics of a software system and specify the commonalities and differences of the products among stakeholders. Features also guide structure, variation, and reuse across all phases of a software system life cycle in product-line engineering [4]. The identification of a *Feature* is inherently quite difficult to capture precisely since the intentions of the stakeholders including end users, design and implementation concepts vary over time. Besides, various domain features become refined and various become obsolete with the valuing of new features [40]. In order to derive a product from a product line using a common set of artifacts, the stakeholders have to decide which features needs to be included and which features needs to be excluded, hence, in

any efficient implementation of software product lines (*SPL*) using *feature-oriented software development (FOSD)* programming paradigm, the underlying source code must comply with the general concept of *Variability* [42]. Over time, different variability modeling approaches have been derived, each with slightly different focus and goal, a very common and popular approach though is to express variability in terms of optional and common features, this approach is often referred to as *feature modeling* [3], and the graphical representation of such models are termed as *feature diagrams* [3]. In a feature model requirements in a domain are expressed at an abstract level. All variable and common properties of products in a product family are described and documented in a feature model. Feature model is composed of a tree structure where each node represents a feature (see Figure 2.1) [20]. A grouping of features represent feature variability, there are mainly two types of feature groups:

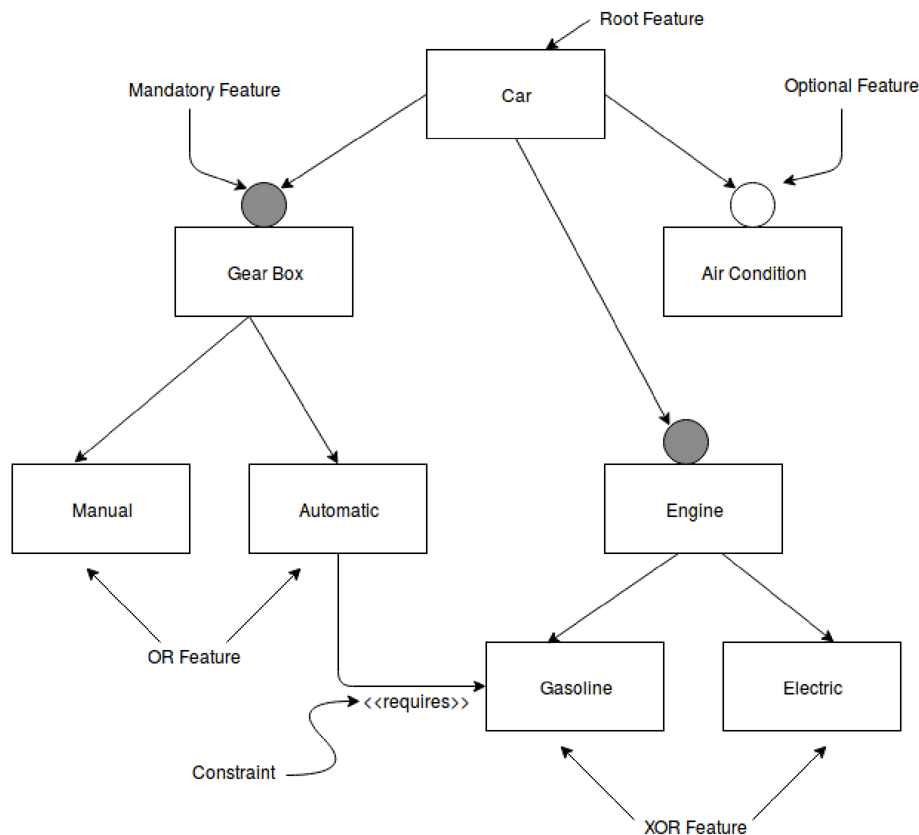


Figure 2.1: Feature model describing a car SPL.

- *Mandatory:* Are those (sub)features that must exist in every product in a product line.

- *Optional:* Are those (sub)features that are optional and thus may be present in some products in a product line.

Besides, the relationship between features, constraints across the nodes of the feature model tree are also practiced to constrain the derivation of a product from a product-line [24]. Following are some of the common types of constraints.

- *And:* When deriving a product from an SPL, all (sub)features must be selected.
- *Xor:* When deriving a product from an SPL, only one (sub)features can be selected.
- *Or:* When deriving a product from an SPL, one or more (sub)features can be selected.
- *Require:* The selection of one (sub)feature requires the selection of another feature.

Figure 2.1 illustrates an example feature model of a Car SPL. Notice, that features "engine" and "gearbox" is mandatory, however, feature "air condition" is optional. Moving further into the tree hierarchy we see that the feature "gearbox" can be either manual or automatic, whereas, the feature "engine" could exclusively either "gasoline" or "electric". Furthermore, if the car has an "automatic" gearbox then it must be installed with a "gasoline" engine [34].

2.2 Variability Implementation Mechanisms

Deriving a product automatically from variable source code, based on user's feature selection is the foremost objective in feature-oriented product line engineering, therefore, different mechanisms have been evolved over the time to implement variability in the source code. These mechanisms can be categorized mainly into two approaches, that is, *composition-based* approach and *annotation-based* approach [22].

2.2.1 Composition Based Approach

In a composition-based approach source code for a specific feature or feature combination is located in a dedicated file, module, or container. In classic composition-based approaches, each feature is developed in the form of a separate plugin which can be plugged directly into a framework, this way various products can be generated via integrating different plug-ins [22]. But besides the classic composition-based approaches, like frameworks and components, there also exists advanced composition-based modular approaches such as *feature-oriented programming (FOP), aspect-oriented programming (AOP)*, and *delta-oriented programming (DOP)*.

In *feature-oriented programming (FOP)* composition-based approaches the design and code of the system are decomposed into the features it provides [16]. The

structure of the whole system aligns with its features, ideally, each feature is built into a separate module or component, hence, new language constructs are required to indicate which elements of a program contribute to which features and to encapsulate each feature source code into a composable, modular unit [5]. Features are mapped to their respective implementations, and therefore, whenever any set of features are selected the corresponding module's implementation is composed automatically [16].

Aspect-oriented programming (AOP) approach is more focused on the modularity of the underlying source code by allowing the separation of *cross-cutting concerns* [13]. It does that by including additional behavior to existing source code without changing the source code itself [13]. Crosscutting usually result in source code scattering and tangling, hence, AOP main aim is to reduce such behaviors (scattering, tangling, and replication) that are prompt by *concerns* which are not well-separated [26].

Delta-oriented programming (DOP) is another approach of implementing variability mechanism in a modular fashion, in which, the program is built of a base module along with a set of delta modules that can alter the base module in a stepwise manner [39]. Delta-oriented programming is also closely related to feature-oriented and aspect-oriented programming [39].

2.2.2 Annotation Based Approach

In the annotation-based approaches source code for all the features in a system is merged into a sole code-base, and annotations indicate which source code implements which feature. That is to say, an annotation is kind of a function which maps program element to its belonging feature or feature combination.

A prominent example of implementing variable source code using annotation-based approaches is via employing *C* preprocessor *cpp*, which is very commonly used in *C* and *C++* based projects [41]. C preprocessor cpp is a lightweight meta-programming tool which supplies program with *directives* which can be employed to modify source file before passing it over to the compiler [23]. It provides the abilities for file inclusion like header files using `#include` directive, defining custom macros using `#define` directive to replace all occurrences of a corresponding token with another token sequence using the preprocessor, and permitting conditional compilation of code based on user-defined conditions using `#if`, `#ifdef` directives [3].

The Macros constructed using `#define` directive can be used as compile-time variables and functions, therefore, during file processing at compile-time they can also be redefined, and undefined. The conditional compilation directives also evaluate compile-time expressions, for example, checking whether macros are defined or not. Consider the simple example fragment of *C* code in Listing 2.1, on line `1` a user-defined header file has been included using the `#include` preprocessing directive, then, at line `2` value of PI has been defined using `#define` directive. Finally, the usage of the conditional directive can be observed from line `3` to `5`, where, if the value of PI is already defined then `#ifdef` changes the value to `3.145`.

In variable software systems programmers annotates conditional parts of the source code using conditional preprocessor directives, such as `#if`, `#ifdef`, `#ifndef`, `#elif`,

```
1 #include "add.h"
2 #define PI 3.14
3 #ifdef PI
4     #define PI 3.145
5 #endif
6 void main()
7 {
8     printf("%f", PI);
9 }
```

Listing 2.1: A Simple C program example, illustrating different preprocessor types.

`#else`, and `#endif` which control lexical program transformations [31]. When a condition that belongs to a conditional directive evaluates to true, only then the conditional code is included. The condition composed of one or more integer based constant and must be defined prior to a conditional directive. Listing 2.2 shows a sample source code example of conditional inclusion, and, Listing 2.3 corresponds to the result after applying *cpp* to it [31]. It can be clearly noticed from the resultant source code (see: Listing 2.3) that the constants `FEAT_SIGNS` and `PROTO` have been set to *true* (or atleast one of them) and the constant `FEAT_NETBEANS_INTG` has been set to *false* (revealed through the exclusion of the statement `struct signlist` × `prev;` on line 12). Each `#ifdef` is responsible for the inclusion of all succeeding lines of code until the next `#ifdef` occurs. Hence, a programmer can easily manage optional source code. Alternative source code snippets are also manageable using the combination of conditional directives (`#if`, `#elif`, and `#else`). Different variant can also be generated out of the box using values of these integer based constants in variable software system, like in our given example, three variants are possible, that is, with `struct signlist` from Line `7` to `15` and/or `struct signlist *prev;` on line `13`, and without both elements [31].

```
1  struct buffblock {
2  struct buffblock *b_next;
3  char_u b_str[1];
4  };
5  #if defined(FEAT_SIGNS) ||
      defined(PROTO)
6  struct signlist {
7  int id;
8  linenr_T lnum;
9  int typenr;
10 struct signlist *next;
11 #ifdef FEAT_NETBEANS_INTG
12 struct signlist *prev;
13 #endif
14 };
15 // some macro definitions
16 #endif
```

Listing 2.2: Conditional Inclusion

```
1  struct buffblock {
2  struct buffblock *b_next;
3  char_u b_str[1];
4  };
5  struct signlist {
6  int id;
7  linenr_T lnum;
8  int typenr;
9  struct signlist *next;
10 };
11 // some macro definitions
```

Listing 2.3: Result after *CPP*

C preprocessor *cpp* remains oblivious to the underlying source code language, therefore, it can be incorporated with many other languages, with syntax similar to *C* language, for example, *C++*, *Java*, *Fortran*, and *assembly* languages [3]. Besides *cpp*, there are several other preprocessors that exist as well and are employed for different purposes.

2.3 Variability-aware Program Analysis

With the emergence of variability management and generator technology individual variants can be derived from the variable code base, provided a set of configuration options. However, variability comes as a trade-off of increased complexity, hence, this trend raised new challenges, that is, the generation of possibly millions/billions of program variants in parallel cannot be analyzed efficiently for errors with classical analysis techniques. A *brute-force* strategy for analyzing all variants in isolation is also not possible. *Sampling heuristics* approach using classic analysis technique address this issue to some extent and reduces the analysis effort significantly. In sampling heuristics approach a sample set of product variants has been picked either with the concern of domain expert or via using an algorithm. Over time, different sampling heuristics approaches have been evolved, for example, *single configuration*, *random*, *code coverage*, and *pair-wise* coverage are some of the most commonly used approaches [17][37]. Having said that, the information acquired using sampling heuristics is still insufficient because it can never be scaled to millions/billions of program variants [32].

In recent times, researchers and practitioners have developed a new class of analyses called *variability-aware* program analysis which analyze the variable code base directly by exploiting the similarities between individual variants and hence reduces analysis effort. In such an approach, the whole variable code base of a variable software system is analyzed before generating any variants, unlike, aforementioned approaches where analysis is been done on individual variants separately [32]. Hence, many variability-aware program analysis tools have been evolved over time, such as, *TypeChef* and *cppstats*. *TypeChef*, parses unpreprocessed *C* source code and encodes the variability in the form of an abstract syntax tree using *presence conditions* [25]. Likewise, *cppstats* also used for analyzing software systems regarding their preprocessor-based variability. Both these aforementioned tools mainly focus on software systems which are explicitly developed in *C* language and are using the capabilities of the *C* preprocessor *cpp* to express variability [30].

2.4 Change Impact Analysis

During the evolution of a software system, maintenance is a key operation. Undesirable side effects and/or ripple effects are typically inevitable to different parts of a software system when a change has been made to a particular software system. *Change Impact Analysis (CIA)* is a way to identify the rippling effects of a change that has been made or to predict the side effects of a proposed change beforehand

[34]. *CIA* is also very handy in circumstances where the stakeholders want to predict the expense and time required to achieve a certain change in a software system. In general, a change impact analysis starts off with a set of changed elements, referred to as *change-set*, and then seeks to determine a much larger set of elements called *impact-set* which requires maintenance effort. [7]. Since the last *30* years, researchers in the domain of software engineering have proposed several definitions and developed numerous change impact analysis (*CIA*) techniques [28]. In general, *CIA* techniques are categorized into *dynamic CIA* and *static CIA* techniques.

2.4.1 Dynamic CIA

Dynamic CIA analysis is performed on the information collected during program execution (e.g, execution traces information, execution related information, and coverage information) to compute the impact set, providing specific inputs. Dynamic analysis techniques are often observed to be more costly, and on top of that, it brings more overhead since the collection of execution information is only possible once we execute the program, thus the ultimate results analysis is also dependent on the program execution [15]. Moreover, Dynamic *CIA* can be executed online or offline. Offline *CIA* is executed after the program completes its execution, whereas online *CIA* executes all analysis via utilizing information accumulated while the program is executing [6].

2.4.2 Static CIA

Static CIA analysis techniques in contrary to dynamic *CIA* analysis techniques are less expensive, since it does not require program execution, but rather, it relies on the analysis of syntax and semantic, or evolutionary dependencies of the program (or its change history repositories) and, therefore, 70% of the existing *CIA* techniques are based on static ones [28]. Static analysis can be further categorized into subcategories, that are, *structural static analysis*, *textual analysis*, and *historical analysis*. In the structural static analysis, structural dependence of the program and the construction of the dependence graph is statically analyzed. Textual analysis works on comments and/or identifiers in the source code, and extracts conceptual dependence (coupling) - based on its analysis. And lastly, the historical analysis explores information from historic revisions of the software system via software repositories [19].

2.4.3 Program Slicing

One of the most efficient technique to perform routine change impact analysis for the latest committed changesets is to use *static program slicing*. It scales expeditiously for programs having few thousand lines of code and serves precise change impact analysis. When performing slicing on a program, one of two kinds of the slice can be constructed: a *forward slice* or a *backward slice* [45]. A *backward slice* contains only those statements of a program which can have some direct or indirect effect on the slicing criterion (variables chosen on the selected point of interest in a program). It

also aids developers in locating parts of a program which contain bugs. Backward slice is sort of a version of the original program, therefore, just like a normal program it can be compiled and executed. One prominent characteristic of the backward slice is, that it maintains the effect of the original program on the chosen variables (selected point of interest in a program) [43].

Consider the program fragment in Listing 2.4 which demonstrate how a backward slice assists in debugging during the development of a program. Notice, that the program is supposed to read exam marks and then supposed to print the number of fails, passed, passing rate, and the average mark.

```
1  Pass = 0;
2  Fail = 0;
3  Count = 0;
4  while (!eof()) {
5  TotalMarks=0;
6  scanf("%d",Marks);
7  if (Marks >= 40)
8  Pass = Pass + 1;
9  if (Marks < 40)
10 Fail = Fail + 1;
11 Count = Count + 1;
12 TotalMarks = TotalMarks+Marks;
13 }
14 printf("Out of %d, %d passed and %d failed\n",Count,Pass,Fail);
15 average = TotalMarks/Count;
16 /* This is the point of interest */
17 printf("The average was %d\n",average);
18 PassRate = Pass/Count*100 ;
19 printf("This is a pass rate of %d\n",PassRate);
```

Listing 2.4: A program fragment to be backward sliced.

However, due to some mysterious error in the program, the average mark always seems to be very low. To find the error, instead of debugging the whole program, an efficient approach is to invoke a backward slice on the program (see Listing 2.5).

```
1 while (!eof()) {
2 TotalMarks=0;
3 scanf("%d",Marks);
4 Count = Count + 1;
5 TotalMarks = TotalMarks+Marks;
6 }
7 average = TotalMarks/Count;
8 printf("The average was %d\n",average);
```

Listing 2.5: The backward slice for variable `average` at line 17 in Listing 2.4.

Notice, that the backward slice constructed for the variable `average` on line `17` (where it is printed) is half the size of the original program, since it only has to preserve the effect of the original program for the variable `average`, thus, makes debugging more simple and convenient. We can clearly notice in the slice (see: Listing 2.5) that the variable `TotalMarks` inside the `while` loop has been assigned

with the value of *zero* which is wrong as this sort initialization usually takes place outside the loop, and therefore, should be initialized at the start of the program or before the loop.

A *forward slice* contains only those statements of a program which are directly or indirectly affected by the slicing criterion (variables chosen on the selected point of interest in a program). It assists in predicting those parts of a program that will be affected by a change. When a program undergoes a change the after-effects of that change 'ripples' through the program and may result in messing with other parts of the program that ought to remain the same. Hence, through forward slicing these ripples can be traced down to track those parts of the program that may have been affected [45].

```
1  n = 0;
2  product = 1;
3  sum = 1;
4  scanf("%d",&x);
5  while (x >= 0){
6  sum = sum + x;
7  product = product * x;
8  n = n + 1;
9  scanf("%d",&x);
10 }
11 average = (sum - 1) / n;
12 printf("The total is %d\n",sum);
13 printf("The product is %d\n",product);
14 printf("The average is %d\n",average);
```

Listing 2.6: A program fragment to be forward sliced.

Consider the program fragment in Listing 2.6 and assume that line 12 which is printing the value of the variable `sum` has just been added to the program. The addition of this line has no rippling effects because it doesn't change any other variables. It does, however, revealed later that a bug exists in the program, that is, the value stored in the variable `sum` is always one more than it should be. To locate the cause for this bug a backward slice has been constructed on the variable `sum` (see: Listing 2.7) on line 12, which revealed that the issue lies in the initialization of the variable `sum` on line 3, that should be `sum = 0;` instead of `sum = 1;`

```
1  sum = 1;
2  scanf("%d",&x) ;
3  while (x >= 0) {
4  sum = sum + x;
5  scanf("%d",&x);
6  }
7  printf("The total is %d\n",sum) ;
```

Listing 2.7: The backward slice for variable `sum` at line 12 in Listing 2.6.

However, changing the assignment of the variable `sum` have some rippling effects for sure, which can be traced through constructing a forward slice. Listing 2.8 indicate the lines (with the comment /*AFFECTED */) of a forward slice. Notice, that

the reassignment of the variable `sum` inside the while loop and the line printing out its value just before the loop ends are affected directly by the change. However, the slice also indicates that the line that assigns a value to the variable `average` and the line printing out its value are affected as well, thus, after fixing the bug by changing the initialization of the variable `sum` on line `3`, would cause a ripple effect as it introduces a bug in the assignment to variable `average`. A quick ugly 'patch' fix is to replace the assignment of variable `average` on line `11` with a new assignment, that is, `average = sum/n;` Finally, a forward slice should be formed for the fresh assignment to variable `average` so as to ensure that no new ripple effects are produced by modifying the assignment to variable `average`.

```
1  n = 0;
2  product = 1;
3  sum = 0;
4  scanf("%d",&x) ;
5  while (x >= 0) {
6  sum = sum + x; /* AFFECTED */
7  product = product * x ;
8  n = n + 1;
9  scanf("%d",&x);
10 }
11 Average = (sum − 1) / n ; /* AFFECTED */
12 printf("The total is %d\n",sum) ; /* AFFECTED */
13 printf("The product is %d\n",product) ;
14 printf("The average is %d\n",average) ; /* AFFECTED */
```

Listing 2.8: The forward slice for variable `sum` at line 3 in Listing 2.6.

3. Concept and Implementation

In this chapter, we highlight in detail our variability-aware change impact analysis approach. We begin with the 'Concept' section - in which we briefly discuss the tools and utilities that we incorporated in our approach. Then, in the following 'Implementation' section, we look briefly into the workflow of our proposed technique from a development perspective.

3.1 Concept

Our main goal is to uplift support for *CIA* in variable software systems which certainly is an area of great practical relevance to product lines. Our variability-aware *CIA* approach is mainly centered on variable software systems which are programmed in *C* language, and thus using *C* preprocessor *cpp* to present variability in the source code. To this end, we incorporated some existing third-party tools and utilities in our approach, such as *srcML*, *srcSlice* and *cppstats* - which facilitates us with efficient analysis of *C* based programs with preprocessor oriented variability. Next, we discuss briefly how these tools work in general and how they assisted us in our *CIA* implementation.

3.1.1 SrcML

SrcML, is a lightweight, highly scalable, robust, multi-language parsing tool which provides an abstract program representation in *XML* format [11]. It is basically a command-line based open-source application which converts a program source code into srcML (XML format) and conversion from srcML format back to the original source code is also possible [11]. SrcML format preserves the entire original text including white-space, comments, special characters. It is sort of a bootstrapped *AST*, which holds information about the syntactic units of the program. Listing 3.1 illustrate an example code snippet of a *C* program file called *test.c* and its corresponding srcML version in Listing 3.2.

If we remove all the elements and attributes from Listing 3.2 we will end up with the exact same original source code in Listing 3.1. That is to say, all other contents shown in Listing 3.2 are just srcML tags, which identify the various syntactic parts of the source code [10]. Notice, that srcML detects that the *test.c* was originally written in *C*, and therefore, it leverages the knowledge of *C* syntax to distinguish between various elements in the source code. For example, it breaks down the preprocessor statement `#include` "rotate.h" into the include directive and file that was included, which are apprehended using the tags `<cpp:include>`, `<cpp:directive>`, and `<cpp:file>`. Note, that the contents of `<cpp:include>` carry the whole include statement, while `<cpp:directive>` carries only include, and `<cpp:file>` carries only the file name. This hierarchical representation carries across all elements, such that all source-code inside a *block* are enclosed within the `<block>` tags, all source-code inside a *function* definition is enclosed within the `<function>` tags, all source-code inside an *expression* is enclosed within the `<expr>` tags etc. Hence, provides with the ease of exploration and manipulation on a syntactic and hierarchical level [10]. The other additional attributes at the start of the srcML file point to the name of the file that has been parsed, its language, and a revision (i.e. the version of the srcML) [11].

```
1  #include "rotate.h"
2  // rotate three values
3  void rotate(int& n1, int& n2, int& n3)
4  {
5    #ifdef NDEBUG
6    if(n1 >20 || n2 >20 || n3 >20) {
7      abort();
8    }
9    #endif
10
11   int tn1 = n1, tn2 = n2, tn3 = n3; // copy original values
12   // move
13   n1 = tn3;
14   n2 = tn1;
15   n3 = tn2;
16 }
```

Listing 3.1: A C program file to be convert to srcML format.

SrcML eases the exploration, manipulation, and analysis of a program as it works on the *programmer-centric* view of the source code instead of a *compiler-centric* one [12]. The conversion process of the source code in both ways is also lossless, that is, there is no chance of losing a single code statement, comments or formatting on the way and thus *100%* round-trip equivalency is guaranteed from source code to srcML and back to the actual source code [12]. One particular reason why we choose srcML in our proposed *CIA* is that, srcML transforms the entire source code, including preprocessor directives, which implies that all *code snippets*, *missing includes* or *libraries* are also converted just as they are to a well-formed srcML, unlike, other tools where *AST* is built after preprocessing [35]. SrcML is extremely fast and efficient compared to a compiler. For instance, it converts a source code into a srcMl format at a speed of *25 KLOCS/sec* [10]. Once the source code is in srcML format,

different XML tools and technologies can be employed for performing different operations, for example, *XPath* and *XQuery* can be employed for better and fast fact extraction, *XSchema* and *RelaxNG* for validation, and *DOM*, *XSLT*, and *SAX* for transformation purposes [11].

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <unit xmlns="http://www.srcML.org/srcML/src" xmlns:cpp="http://www.
       srcML.org/srcML/cpp" revision="0.9.5" language="C" filename="test.
       c"><cpp:include>#<cpp:directive>include</cpp:directive> <cpp:file>
       "rotate.h"</cpp:file></cpp:include>
3  <comment type="line">// rotate three values</comment>
4  <function><type><name>void</name></type> <name>rotate</name><
       parameter_list>(<parameter><decl><type><name>int</name><modifier>&
       amp;</modifier></type> <name>n1</name></decl></parameter>, <
       parameter><decl><type><name>int</name><modifier>&amp;</modifier></
       type> <name>n2</name></decl></parameter>, <parameter><decl><type><
       name>int</name><modifier>&amp;</modifier></type> <name>n3</name></
       decl></parameter>)</parameter_list>
5  <block>{
6    <cpp:ifdef>#<cpp:directive>ifdef</cpp:directive> <name>NDEBUG</name
         ></cpp:ifdef>
7    <if>if<condition>(<expr><name>n1</name> <operator>&gt;</operator><
         literal type="number">20</literal> <operator>||</operator> <name
         >n2</name> <operator>&gt;</operator><literal type="number">20</
         literal> <operator>||</operator> <name>n3</name> <operator>&gt
         ;</operator><literal type="number">20</literal></expr>)</
         condition><then> <block>{
8      <expr_stmt><expr><call><name>abort</name><argument_list>()</
           argument_list></call></expr>;</expr_stmt>
9    }</block></then></if>
10   <cpp:endif>#<cpp:directive>endif</cpp:directive></cpp:endif>
11
12   <decl_stmt><decl><type><name>int</name></type> <name>tn1</name> <
         init>= <expr><name>n1</name></expr></init></decl>, <decl><type
         ref="prev"/><name>tn2</name> <init>= <expr><name>n2</name></expr
         ></init></decl>, <decl><type ref="prev"/><name>tn3</name> <init
         >= <expr><name>n3</name></expr></init></decl>;</decl_stmt> <
         comment type="line">// copy original values</comment>
13   <comment type="line">// move</comment>
14   <expr_stmt><expr><name>n1</name> <operator>=</operator> <name>tn3</
         name></expr>;</expr_stmt>
15   <expr_stmt><expr><name>n2</name> <operator>=</operator> <name>tn1</
         name></expr>;</expr_stmt>
16   <expr_stmt><expr><name>n3</name> <operator>=</operator> <name>tn2</
         name></expr>;</expr_stmt>
17 }</block></function>
18 </unit>
```

Listing 3.2: A C++ program fragment after applying srcML.

In short, srcML can be utilized to cope with a variety of maintenance problems. For example, the analysis of big software systems to automatically reverse engineer class and method stereotypes, applying transformations to support API and compiler migration, supporting syntactic differencing, and etc. SrcML parser currently

supports *C/C++*, *C#*, and *Java* programs. There are also several tools that are based on the srcML platform and are used for different purposes,[1] for instance:

- *srcPtr:* Is a lightweight tool, used for pointer analysis in the source code.
- *srcSlice:* Is a lightweight and fast, forward static slicing tool.
- *srcType:* Is another lightweight tool, used for static type resolution.
- *srcUML:* Is used for reverse engineer accurate UML class diagrams from code.

3.1.2 SrcSlice

SrcSlice, is an efficient lightweight *forward static slicing* utility tool which is developed on top of the srcML platform (an XML representation of source code). The final result produce includes a list of line numbers, dependent variables, aliases, and function calls that are part of the slice for a given variable. SrcSlice is very scalable and can produce the slices for all variables of the *Linux kernel* in just *15* minutes [36]. In our *CIA* implementation, we make use of this tool to produce the slice for all variables in a system. As a specialty, srcSlice computes the dependence and control information as needed, thus, does not compute the program/system dependence graph [1]. SrcSlice takes as input a srcML (XML based) representation of the source code file. Listing 3.3 illustrates a sample *C++* program file called *test.cpp* and its corresponding forward static slice constructed using srcSlice is given in Listing 3.4.

```
1  int fun(int z){
2   z++;
3   return z;
4  }
5  void foo(int &x, int *y){
6   fun(x);
7   y++;
8  }
9  int main(){
10  int abc = 1;
11  int i = 1 + abc;
12  while (i<=10){
13   foo(abc, &i);
14  }
15  std::cout<<"i: "<<i<<"abc: "<<abc<<std::endl;
16  std::cout<<fun(i);
17 }
```

Listing 3.3: A sample C++ program file to be SrcSlice

[1] https://www.srcml.org/

Note that in the output from srcSlice (see: Listing 3.4), each line corresponds to an individual slice profile. For clarity, let's analyze the first line from the output as a representative slice profile and break down all its corresponding elements moving from left to right to see what each of them is actually meant for:

```
1 test.cpp,main,i,def{11},use{1,2,3,5,7,12,13,15,16},dvars{},pointers
      {},cfuncs{fun{1},foo{2}}
2 test.cpp,main,abc,def{10},use{1,2,3,5,6,11,13,15},dvars{i},pointers
      {},cfuncs{fun{1},foo{1}}
3 test.cpp,fun,z,def{1},use{2,3},dvars{},pointers{},cfuncs{}
4 test.cpp,foo,y,def{5},use{7},dvars{},pointers{i},cfuncs{}
5 test.cpp,foo,x,def{5},use{1,2,3,6},dvars{},pointers{abc},cfuncs{fun
      {1}}
```

Listing 3.4: Slices produced after performing srcSlice on program in Listing 3.3

- `test.cpp,` points to the name of the `file`, that has been sliced.
- `main,` points to the name of the `function`, containing the slicing criterion (slice variable).
- `i,` points to the name of the slice `variable` itself.
- `def{},` also called `index`. Points to a list of line numbers, where the slice variable has been declared or initialized in the function.
- `use{},` also called `slines`. Points to a list of line numbers that comprise the slice, or in other words, a list of line numbers where the slice variable value has been read.
- `dvars{},` also called `dvariables`. Points to a list of variables that are data dependent on the slice variable, or in other words, a list of variables in the program that have been impacted by the slice variable.
- `pointers{},` points to a list of aliases (pointer) of the slicing variable.
- `cfuncs{fun{1},foo{1}},` also called `cfunctions`. Point to a list of functions called using the slicing variable.

As we see the above-mentioned information does not represent the actual slice but rather the slice related information, which we utilize during the implementation of our proposed *CIA* technique. Once a program file undergoes through a srcSlice operation it provides us with all possible slice profiles with respect to each existing

variable in the file. During our implementation phase, we used these slice profiles to track down all possible impacted parts in a program file - given a set of line numbers of modified statements in the file. For instance, consider the program in Listing 3.3 and suppose a change has taken place on line `11`, so we look through all the slice profiles given in Listing 3.4 and checks if `11` exist in the `def{}` list element of any slice profile. Observe, that `11` exist only in the `def{}` element of the first slice profile, therefore, we will then consider all line numbers in the `use{}` list element of the respective slice profile as potentially *impacted* line numbers.

3.1.3 cppstats

Most complex software systems today are configurable and conditional compilation is the most common variability-implementation mechanism that is broadly used in open-source projects especially *C* preprocessor (*cpp*) [18]. *cppstats*, is a powerful tool for analyzing a *C* based software system with regard to its preprocessor-based variability i.e. cppstats, provides information about the usage of the *CPP* in *C* software projects [30]. In a nutshell, cppstats computes the usage of *CPP* directives directly on the normalized version (i.e. the source code without any sort of spacings, comments, and indentations) of the source code. Interestingly, to perform the analysis cppstats itself also leverages the afore-discussed *srcML* tool i.e. to conduct the analysis, the normalized source-code is first transformed to srcML format. All sorts of available analysis in cppstats carry out mainly two steps in general [29] i.e.

- *In the first step,* it normalizes and prepares the source code using the appropriate methodology, and, then converts the *C* source code to srcML format - including *CPP* annotations.

- *In the second step,* it performs any of the below mentioned analyses, based on the abstract srcML representation.

cppstats is a suite of analyses. and therefore comes with different sorts of analysis such as *General*, *GeneralValues*, *Discipline*, *FeatureLocations*, *Derivative*, and *Interaction* [31]. In particular, we are interested in *FeatureLocations* analysis which is basically used for analyzing the locations of *CPP* annotated code blocks in the provided file or project folder. A *FeatureLocations* analysis returns the output in the form of two of tables,[2] which are:

- `cppstats_featurelocations` with the following colums.

  - `FILENAME:` Refers to the name of the file.

[2] http://fosd.net/cppstats

  - LINE_START: Refers to the start line number of the cpp annotated block.
  - LINE_END: Refers to the end line number of the cpp annotated block.
  - TYPE: Refers to the type of the annotation (#if, #elif, #else).
  - EXPRESSION: Refers to the #ifdef expression (involved configuration constants or features).
  - CONSTANTS: Refers to the name of the configuration constant or feature.

- listoffeatures with the following colums.
  - FILENAME: Refers to the name of the file.
  - CONSTANTS: Contains the preprocessor variables used in the expression.

```
1  #ifdef F1
2   #define EXPR (a<0)
3  #else
4   #define EXPR 0
5  #endif
6
7  int r;
8  int foo(int a #ifdef F2 , int b #endif) {
9   if (EXPR) {
10   return -b;
11  }
12  int c = a;
13  if (c) {
14   c += a;
15   #if defined(F1) && defined(F2)
16    c += b;
17   #endif
18  }
19  return c;
20 }
```

Listing 3.5: Sample source file illustrating code to be analyzed with *Featurelocation*.

The table cppstats_featurelocations provides us with some useful stats that we incorporate in the later phase (see: Section 3.2.5) of our main analysis. We clarify this in detail with the help of an example source file called *test.c*. Listing 3.5 shows the source code of *test.c* file and its corresponding cppstats_featurelocations output table after executing *FeatureLocations* analysis on it is illustrated in Table 3.1.

FILENAME	LINE_END	LINE_START	TYPE	EXPRESSION	CONSTANTS
test.c	1	3	#if	defined(F1)	F1
test.c	3	5	#else	(!(defined(F1)))	F1
test.c	8	8	#if	defined(F2)	F2
test.c	15	17	#if	((defined(F1)) && (defined(F2)))	F1;F2

Table 3.1: `cppstats_featurelocations` table with sample data.

Note that the each row in the `cppstats_featurelocations` table (see: Table 3.1) elaborates information about each specific annotated code block in *test.c* file. For instance, observe the data given in the second row of `cppstats_featurelocations` table, here it specifies that the file `test.c` has an annotated code block which starts from `line 1` of the `test.c` file and ends on `line 3` of the `test.c` file. It also specifies the type of the preprocessor directive (`#if`) and the expression (`defined(F1)`) that has been used to annotate the respective code block, along with the involved configuration constant or feature (`F1`) (see: Listing 3.5). This information is quite useful for us during the implementation phase to identify the impacted features i.e. once we compute all affected line numbers in a file (see: Section 3.2.5.1), we can then check them against the `LINE_END` and `LINE_START` columns of the `cppstats_featurelocations` table (and see whether the affected line number lies between the range of the columns values) - provided that the file name also match with the `FILENAME` column value. For example, let suppose couple of changes are been made in `test.c` file on line 14 & 16 in Listing 3.5. The list of affected line numbers will then contains `{14, 16}` so if we check these line numbers in the source file we can clearly see that the line `16` falls in-between the range of `LINE_END` and `LINE_START` columns values (on the fourth row of `cppstats_featurelocations` table in Table 3.1). Thus, from here we can conclude that the features `F1` and `F2` are both impacted by the change.

FILENAME	CONSTANTS
test1.c	F1;F2
test2.c	F1;F3;
test3.c	F1;F2;F3

Table 3.2: `listoffeatures` table with sample data.

Similarly, the second table `listoffeatures` also provides us with some valuable information that we incorporate into our system during the final report generation. We clarify this with the help of an example `listoffeatures` table as illustrated in the Table 3.2 - which we populated with some sample data. Note that each data row in the table (see: Table 3.2) shows the name of the file in the first column and then, the names of all the features that are included in that file, in the second column. Thus, with `listoffeatures` table we can find the list of features that are included in each file of a commit (which undergoes through the cppstats analysis) and consequently use this information to calculate the number of all distinct features that are involved in a particular commit.

3.2 Implementation

Now that we have looked briefly into the major tools that we incorporated in our *CIA* technique, we head over to the general idea of our practical implementation. Our *CIA* implementation is mainly programmed in *Java*, all the source code and APIs are therefore programmed in core Java programming language and the third-party libraries that we incorporated are mostly Java-based as well. We also provide in our system a global configuration file called `config.properties`, through which a user can configure some global system level configurations that are required by the system as input to run any *change impact analysis*. These configuration options includes:

- `gitRepo:` Refers to the *Git* repository URL of the project to be analyzed (mandatory).

- `downloadPath:` Points to the local downloaded path of the project (mandatory).

- `pathToSrc:` Points to the main source code directory path of the locally downloaded project (mandatory). `pathToSrc` value is required by *cppstats* in order to perform any type of above-mentioned analysis. Hence, during the execution of our *change impact analysis*, the system will programmatically writes this value to the configuration file of cppstats called `cppstats_input.txt`.

- `pathToCppstats:` Points to the *cppstats* installation directory path (mandatory). `pathToCppstats` value is required because *cppstats* only executes from the root of its installation directory where the `cppstats_input.txt` configuration file is located. Hence, during the execution of our *change impact analysis* before executing cppstats based operations, the system will programmatically switch path to the cppstats installed directory.

- `oldCommit:` Points to git hash string for the old commit (mandatory).

- `newCommit:` Points to git hash string for the latest commit. If empty, then HEAD will be considered (optional).

Figure 3.1 gives a schematic view of our implementation. The dotted rectangular blocks in the figure points to different implementation phases, which we will discuss briefly in the following sections.

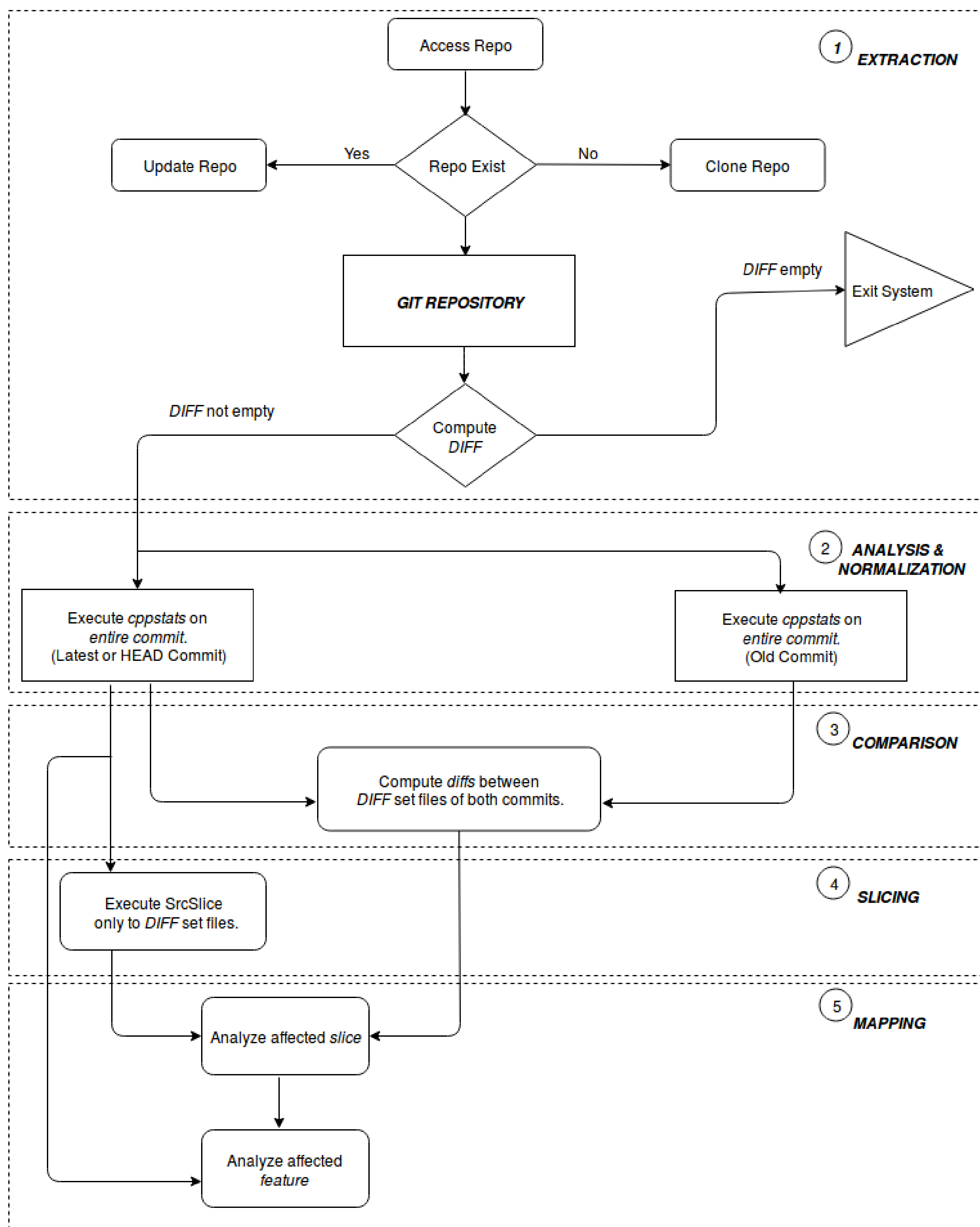


Figure 3.1: Illustration of our proposed CIA implementation.

3.2.1 Extraction

In the first *Extraction* phase of our implemetation, we take as input a *C* based project, which must be cloned from a remote git repository of the respective project, specified by the user through the global configuration file. In case, the project already exists locally, our system will then invoke an update call to the remote git repository (to pull the latest updates). After cloning/updating, we determine a set of files which have been modified between the two commits (specified by the user through the global configuration file) of the project e.g. two subsequent releases. We discover this set of modified files using git utility *APIs* called `git-diff` and refer it to as a *DIFF* set. In order to run our proposed *CIA* successfully a non-empty *DIFF* set is a must required by our system since it leads to those source files in the subject project which have been modified between the two considered commits. Hence, in case the *DIFF* set is empty we halt any further execution of our *CIA*, and the system will exit from there. We perform all these *git* specific operations using *JGit* library which is a full-featured implementation of git, written natively in Java.

3.2.2 Analysis & Normalization

In *Analysis & Normalization* phase, we first execute *cppstats* on the entire latest commit which first transforms all the source files present in the latest commit into an abstract program representation in the form of *srcML* format, and then, undergoes a cppstats based analysis called *FeatureLocations* (see: Section 3.1.3). The information we acquired from the output of executing FeatureLocation analysis on the latest commit helps us in locating all code blocks in each respective source file of the latest commit that are annotated with *CPP* based conditional compilation directives (`#if`, `#elif`, `#else`). Also, from the output result, we compute the list of all the *Features* that are involved in the latest commit of the subject project (see: Section 3.1.3). As explained earlier cppstats normalizes the source files before transforming it to srcML format (see: Section 3.1.3), which lead us to a big hurdle i.e. the files from the latest commit are not in-sync anymore with the source files present in the older commit. Thus, in the following phase (see: Section 3.2.3), where we have to compare each file in the latest commit with its corresponding file in the older commit to compute the exact line numbers which were modified due to the changes becomes impossible. Hence, to address this issue, we run cppstats again on the entire older commit as well. The second cppstats run also aid us in computing the list of all the *Features* that are involved in the older commit of the subject project.

3.2.3 Comparison

In the *Comparison* phase of our implementation, we compare the content of the files (that belong to the *DIFF* set) of the two considered commits. For that purpose, we make use of a Java-based open-source library called *DiffUtils* [3] which computes the *diff* (line numbers of the changes in a source file) between the source files of the two considered commits, and as a result, provides with a list of line numbers for all the

[3]https://code.google.com/archive/p/java-diff-utils/

changes in each file. DiffUtils library in general implements *Myer's diff algorithm* for computing diffs. The *diffs* can be categorized into *modification*, *insertion* and *deletion*.[4]

- *Modifications:* Refers to all the changes that take place in the newer revision of a file with respect to the older revision of the file.
- *Insertion:* Refers to all the new statements that have been added in the newer revision of a file with respect to the older revision of the file.
- *Deletion:* Refers to all the statements that have been removed in the newer revision of a file with respect to the older revision of the file.

For simplicity, consider a source file with two different revisions as illustrated in Listing 3.6 and Listing 3.7, the *diff* list for the given example after invoking DiffUtils library will contain the following line numbers `2`, `3`, `4`, `11` and `12`.

```
1  int foo(int x, int y){
2   return 0;
3  }
4  int test(){
5   int x;
6   x = 1;
7   int y;
8   y = x + 1;
9   foo(x, y);
10  std::cout<<y;
11 }
```

Listing 3.6: Old Revision

```
1  int foo(int x, int y){
2   int z;
3   z = x + y;
4   return z;
5  }
6  int test(){
7   int x;
8   x = 1;
9   int y;
10  y = x + 1;
11  foo(x, y);
12 }
```

Listing 3.7: New Revision

While comparing each file from the latest commit to its corresponding file from the older commit, we only consider the *diff* lines which belongs to *insertions* and *modification* types. The reason for discarding the *diff* lines which belongs to *deletion* type is that we are only interested in the *diff* lines which exist in the latest version of the file (i.e. from the latest commit). To this end, we create a list of *diff* lines for each file, which we refer as `diffLines{}`. Hence, at the end of the comparison phase, we end up with a `(key, value)` pair map list output called `filediffs{}` - which contain a list `L` (consist of `diffLines{}` element) as `value`, and the name of the respective file `F` as the `key`.

[4] https://code.google.com/archive/p/java-diff-utils/

3.2.4 Slicing

In the *slicing* phase we constructs *forward static* slices for each file in the *DIFF* set (from the latest commit only) using the *srcML* platform based tool called *srcSlice*. The srcSlice tool uses the same *srcML* files as inputs (only the *DIFF* set related files), which were generated during the first run of cppstats over the latest commit. This provides us with all possible *slice profiles*, for each existing variable in a source file. Each slice profile provides with a bunch of information with respect to its slice variable. But as described earlier, we are only interested in the `def{}` and `use{}` elements of each slice profile (see: Section 3.1.2). To this end, we crop the slice profiles and extract only the `def{}` and `use{}` elements from each slice profile and store it into a list called `sliceProfiles`. Hence, the final output of the slicing phase contains a `(key, value)` pair map list called `fileSlices{}` - which contain a list `L` (consist of `sliceProfiles{}` element) as `value`, and the name of the respective file `F` as the `key`. Algorithm 1 gives a pseudo representation of the slicing phase.

Algorithm 1

INPUT: `files`
OUTPUT: `fileSlices`

```
 1: for all file ∈ files | file ∈ DIFF & file ∈ Latest commit do
 2:   slices ← execute.srcSlice(file)
 3:   for each slice ∈ slices do
 4:     slice ← cropSlice(slice)
 5:     sliceProfiles[] ← slice.def; slice.use
 6:   end for
 7:   fileSlices(key) ← file.name
 8:   fileSlices(value) ← sliceProfiles
 9: end for
10: return fileSlices
```

3.2.5 Mapping

The final *Mapping* phase is where all the main analysis work has been performed using the data that we collected from the previous phases. The mapping phase is mainly composed of two analyses. The first analysis we termed as *analyze affected slice (AAS)* - in which we compute all the affected slice profiles with respect to the subject file. The second analysis we termed as *analyze affected feature (AAF)* - in which we compute all the affected features.

3.2.5.1 Analyze affected slice

For the execution of the *AAS* analysis we require two inputs. The first input is the `(key, value)` pair map list i.e. `filediffs{}` - which we received as an output from the *comparison* phase. The second input is also a `(key, value)` pair map list i.e. `fileSlices{}` - which we received as an output from the *slicing* phase.

Algorithm 2

INPUT: `filediffs`, `fileSlices`
OUTPUT: `affectedFiles`

```
 1: for all entry ∈ filediffs do
 2:    file ← entry.key
 3:    diffLines ← entry.value
 4:    for all entry ∈ fileSlices | entry.key = file do
 5:       sliceProfiles ← entry.value
 6:       for each diffLine ∈ diffLines do
 7:          for each slice ∈ sliceProfiles do
 8:             defLines ← slice.def
 9:             for each defLine ∈ defLines do
10:                if defLine = diffLine then
11:                   affectedLines[] ← defLine
12:                   if slice.use ≠ NULL then
13:                      useLines ← slice.use
14:                      for each useLine ∈ useLines do
15:                         affectedLines[] ← useLine
16:                      end for
17:                   end if
18:                end if
19:             end for
20:          end for
21:       end for
22:    end for
23:    affectedFiles(key) ← file.
24:    affectedFiles(value) ← affectedLines.
25: end for
26: return affectedFiles
```

We first take a `value` from the `filediffs` map list (`diffLines`), and a `value` from the `fileSlices` map list (`sliceProfiles`) - provided that the `keys` of both the values from there respective map list must be equal. Then, for each line number in `diffLines` list we compare it with each line number in the `def` sublist inside `sliceProfiles` list value (each `sliceProfiles` list value is composed of two sublists i.e. `def` and `use` see: Section 3.2.4). If a line number, let suppose `x` from the `diffLines` matches with a line number `x` in the `def`, it means that the respective slice profile has been affected by the change. Hence, we consider all the line numbers in the `use` sublist (second sublist inside each `sliceProfiles` list value) as well as the line number `x` itself (see: Section 3.1.2), and store it into a new list element called `affectedLines`. We repeat the process for all the entries in the `filediffs` map list, and eventually end up with a `(key, value)` pair map list called, `affectedFiles{}` - in which each `key` corresponds to the `key`(filename) of the `filediffs`, and each `value` corresponds to the respective `affectedLines{}` list element. Algorithm 2 gives a pseudo representation of the *AAS* analysis.

3.2.5.2 Analyze affected feature

After the completion of *AAS* analysis, we head over to our final *AAF* analysis in which we figure out all those features in the system which may have been impacted by the changes, and, as a result generate different reports in the form of tables. The *AAF* analysis requires two inputs, the first input is the `(key, value)` pair map list i.e. `affectedFiles{}` - which we received as an output from the *AAS* analysis. The second input is based on the resultant `cppstats_featurelocations` table - which was created during the first run of *cppstats* over the entire latest commit in the *Analysis & Normalization* phase (see: Section 3.2.2). Algorithm 3 gives a pseudo representation of the *AAF* analysis.

Algorithm 3

INPUT: `affectedFiles`, `cppstats_featurelocations`
OUTPUT: `ciaStats`

```
1: Table ciaStats = new Table.columns(FILENAME,DIFFLINE,
              CPPANNOTATION_STARTLINE,CPPANNOTATION_ENDLINE,
              TYPE,EXPRESSION)
2: for all entry ∈ affectedFiles do
3:    file ← entry.key
4:    affectedLines ← entry.value
5:    dataRows ← Select * from cppstats_featurelocations where
              cppstats_featurelocations.FILENAME = file
6:    for each affectedLine ∈ affectedLines do
7:       for each row ∈ dataRows do
8:          if affectedLine ≥ row.LINE_START &
                affectedLine ≤ row.LINE_END then
9:             Insert into ciaStats set ciaStats.FILENAME =row.FILENAME,
                  ciaStats.DIFFLINE = affectedLine,
                  ciaStats.LINE_START = row.LINE_START,
                  ciaStats.LINE_END = row.LINE_END,
                  ciaStats.TYPE = row.TYPE,
                  ciaStats.EXPRESSION = row.EXPRESSION
10:         end if
11:      end for
12:   end for
13: end for
14: return ciaStats
```

First, we declare a table called `ciaStats` which we use to store the output data, we will going to acquire from our *AAF* analysis. Next, we loop through `affectedFiles` map list and then for each `key` (filename) from the `affectedFiles` map list, we retrieve all the rows from the `cppstats_featurelocations` table - provided that the first column (`FILENAME`) values from `cppstats_featurelocations` must match with the respective `key` of `affectedFiles`. Then, we take the `value` from the `affectedFiles` map list, which is a list element (`affectedline{}`) and for each line

number in that list element, we check whether that number exist between the range of `LINE_START` and `LINE_END` columns values of the `cppstats_featurelocations` table rows (which we retrived before). If the line exists, we conclude from this that the respective annotated code block is impacted by a change. Hence, the *configuration constant* or *feature* involved in that annotated code block is also considered as an affected feature (see: Section 3.1.3). To this end, we store the following information from each `true` iteration into our pre-declared table (`ciaStats`) and repeat the process for all the entries in the `affectedFiles` map list.

- `FILENAME:` Refers to the name of the file.
- `DIFFLINE:` Refers to the changed line number i.e. `affectedline`.
- `LINE_START:` Refers to the start line number of the cpp annotated block.
- `LINE_END:` Refers to the end line number of the cpp annotated block.
- `TYPE:` Refers to the type of the annotation (`#if`, `#elif`, `#else`).
- `EXPRESSION:` Refers to the `#ifdef` expression (involved configuration constants or features).

Finally, after doing some computations on the data we collected through the aforementioned `ciaStats` table, we create a couple of additional tables i.e. `FeatureCounts` and `FeatureStats` - which provide the end-user of our system with more insight regarding the features involved in the two considered commits with respect to change impact.

- Table `FeatureCounts` includes the following columns.
  - `Feature:` Name of each *feature* involved in the latest commit.
  - `Frequency:` Number of time the corresponding *feature* has been impacted by the change.
- Table `FeatureStats` includes the following columns.
  - `NFL:` Number of *features* involved in the latest commit.

- `NFO`: Number of *features* involved in the old commit.
- `NCF`: Number of *features* that are common between old and latest commit.
- `NIF`: Number of total impacted *features*.
- `NCIF`: Number of common impacted *features* out of the total impacted *features*.

4. Evaluation

In this chapter, we briefly highlight the results of evaluating our proposed approach regarding its benefit and performance. We kickoff with our research questions in the first section of this chapter, then in the following section we discuss briefly the subject systems that we used as a test case for evaluating our approach. Next, we introduce methodology i.e. how we processed with our testing in general. Finally, in the last section of the chapter, we briefly conclude the results of our evaluations.

4.1 Research Questions

Our evaluation mainly investigates the performance of our proposed *CIA* approach regarding the impact of changes to the features that exist in variable software systems which consequently can serve in the development and maintenance of a product line in domain engineering, i.e. determining the different product variants affected by the change. In particular, we used some real-world opensource variable software systems in pursuit to explore the following research questions:

- **RQ1.** *How well our variability-aware CIA approach determines the affected features in a variable software system?* Our approach very competently specifies the set of all impacted features between the comparison of two considered commits. We thus verify this via performing our *CIA* between latest commits and then verify the results against the stats we gathered manually.

- **RQ2.** *How many common features are impacted between two major releases in a variable software system?* It is quite certain that between any two commits (major releases) there are features which get affected by the changes, and therefore, our variability-aware *CIA* not only indicate the set of affected features with respect to the latest commit but also pinpoint the set of common features affected between the two considered commits.

- **RQ3.** *Which features, in general, are more frequently impacted by changes to a variable software system?* In a highly configurable system where there are many features involved some particular features, in general, get affected in every other commit (release). Hence, our variability-aware *CIA* also efficiently estimate the features which are more often affected by the changes.

4.2 Subject Systems

To investigate the research questions that we highlighted in the previous section, we evaluate our proposed tool with some real-world open-source systems, that are, *OpenVPN*, *BusyBox*, and *Vim*. Next, we will briefly explain each of these systems, their domain, and some meta information.

4.2.1 OpenVPN

OpenVPN is a robust and highly flexible open source VPN daemon mainly programmed in *C* language, developed by James Yonan. In the pursuit to become a universal level VPN tool, OpenVPN supports several features such as SSL/TLS security, TCP or UDP tunnel transport through proxies or NAT, ethernet bridging, support for dynamic IP addresses and DHCP, with scalability to up to hundreds or thousands of users, and portability to most major OS platforms. OpenVPN is tightly connected to the OpenSSL library and reaps much of its crypto capabilities from it [14]. OpenVPN provides conventional encryption via utilizing a pre-shared secret key (Static Key mode) or public key security (SSL/TLS mode) using client and server certificates. OpenVPN also provides non-encrypted TCP/UDP tunnels and is designed to operate with the TUN/TAP virtual networking interface that exists on most platforms.[1]

4.2.2 BusyBox

BusyBox is a software that gives various stripped-down Unix tools in a sole executable file and provides alternatives for most of the utilities user find in GNU fileutils, shellutils, etc (though, the busybox provided utilities have fewer options compared to the full-featured GNU utilities). BusyBox can execute in a family of POSIX environments, for instance, Linux, FreeBSD, and Android even-though most of the tools it provides are meant to work with interfaces given by the Linux kernel [46]. In general, busybox provides with a reasonably complete environment for a small or embedded system. BusyBox has been natively written in *C* language considering the fact of size-optimization and limited resources in mind. It is also notably modular so the user can readily include or exclude commands (or features) at compile time, and thus, customize the embedded systems.[2]

[1] https://openvpn.net/
[2] https://busybox.net/

4.2.3 Vim

Vim is a highly configurable and stable text editor for efficiently writing and altering any kind of text and is an exceedingly improved version of the good old UNIX editor Vi [38]. Vim is incorporated as ”vi” with most UNIX systems and with Apple OS X. Vim can also run under MS-Windows (NT, 2000, XP, Vista, 7, 8, 10). Vim offers various tremendous features, such as persistent multi-level undo tree, extensive plugin system, syntax highlighting, command line history, on-line help, spell checking, support for hundreds of programming languages and file formats, powerful search and replace, integrates with many tools, and etc. Vim also provides with a Graphical User Interface (GUI).[3]

Table 4.1 represents the most latest (at the time of writing) meta data information of the subject systems and each column in the table corresponds to the following information.

- *LOC* - refers to the total number of *lines of code* involved in the subject system.
- *LOF* - refers to the total number of *lines of CPP-annotated code* involved in the subject system.
- *NOFC* - refers to the total *number of feature constants* involved in the subject system.
- *NOC* - refers to the *number of commits* as per the official *git* repository history stats of the subject system.
- *NOR* - refers to the *number of releases* as per the official *git* repository history stats of the subject system.

	LOC	LOF	NOFC	NOC	NOR
OpenVPN	70455	39588	470	2308	76
BusyBox	170682	48027	1537	15698	143
Vim	340776	194230	1110	8975	7434

Table 4.1: Latest meta information of the subject systems

[3]https://www.vim.org/

4.3 Methodology

As a baseline for our evaluation, we selected several commits from each selected software system so that we can better investigate the research questions we highlighted in the first section. Moreover, all the selected commits from each subject system correspond to a major stable release version. Table 4.2, Table 4.3, and Table 4.4 present the list of selected commits (major releases) from the *OpenVPN*, *BusyBox*, and *Vim* subject systems respectively where each row in the given tables corresponds to the metadata information of a particular selected commit.

Commits	Release dates	LOC	LOF	NOFC
v2.4.6	04/19/2018	69932	46559	476
v2.4.5	02/28/2018	69926	46555	476
v2.4.4	09/25/2017	69621	46287	451
v2.4.3	06/20/2017	69644	46273	453
v2.4.2	05/11/2017	69232	45931	434
v2.4.1	03/21/2017	69019	45793	434
v2.4.0	12/27/2016	68528	45437	420
v2.3.5	10/25/2014	54777	37698	402
v2.3.4	04/30/2014	54695	37641	401
v2.3.3	04/08/2014	54614	37565	401
v2.3.2	05/31/2013	54249	37259	394
v2.3.1	03/27/2013	54236	37250	395
v2.3.0	01/07/2013	54028	37027	394

Table 4.2: Meta data of the *OpenVPN* selected commits

From each subject system, first, we selected two latest release commits and perform our variability-aware *CIA* between them, which help us in investigating the set of impacted features in a shorter period of time. Then, we selected some more release commits which are around *1 to 2* years older than the most latest release commit and will perform our variability-aware *CIA* between them. This way we have a deeper insight into how different features are evolving over time. Finally, we selected a few more release commits which are around *3 to 8* years older than the most latest release commit and perform our variability-aware *CIA* between them. This again helps us better understand the set of impacted features over a longer period of time and also exposing the features which are more frequently affected by the changes. Note, in the tables (see: Table 4.2, Table 4.3, and Table 4.4) that the number of *LOC* (lines of code) and the number of *LOF* (lines of *CPP*-annotated code) in each table is highest for the most latest commit and as we go down (by date) in the tables to the older commits it decreases. Similarly, the number of *NOFC* (number of feature

constants) is also highest for the most latest commit and gradually drops down as we go down (by date) in the tables to the older commits. It is also worth noticing that in *OpenVPN* and *BusyBox* subject systems the number of feature constants (*NOFC*) involved in the considered commits ranges between *300* to *500*. However, in the *Vim* subject system, the number of feature constants (*NOFC*) involved in the considered commits are relatively higher i.e. ranges from *900* to *1110* - which intimates that *Vim* is a much more variable system compare to the other two subject systems.

Commits	Release dates	LOC	LOF	NOFC
v1.29.3	09/09/2018	170047	47790	1531
v1.29.2	07/31/2018	170046	47790	1531
v1.29.0	07/01/2018	170041	47789	1531
v1.28.0	01/02/2018	168204	46399	1506
v1.27.0	07/03/2017	166023	45508	1490
v1.26.0	12/20/2016	159961	41157	1400
v1.25.0	06/22/2016	159169	40566	1370
v1.24.0	10/12/2015	158023	40116	1352
v1.23.0	12/24/2014	184266	41815	1434
v1.22.0	01/01/2014	181948	40758	1408
v1.21.0	01/21/2013	180977	40466	1401
v1.20.0	04/22/2012	180097	40271	1392
v1.19.0	08/13/2011	178183	39796	1383
v1.18.0	11/23/2010	176420	39025	1383

Table 4.3: Meta data of the *BusyBox* selected commits

4.3.1 Evaluation Metrics

The execution of our variability-aware *CIA* on the selected commits provides with some valuable information regarding the subject systems development/maintenance life cycle, which we structure into a resultant matrix as described below.

- `NFL` stands for the total number of *features* involved in the latest commit out of the two compared commits.

- `NFO` stands for the total number of *features* involved in the old commit out of the two compared commits.

- `NCF` stands for the total number of common *features* i.e. `NFL` ∩ `NFO`.

- `NIF` stands for the total number of impacted *features* in the latest commit.

- `NCIF` stands for the total number of common impacted *features* i.e. `NCF` ∩ `NIF`.

Commits	Release dates	LOC	LOF	NOFC
v8.1.0488	10/20/2018	340776	194230	1110
v8.1.0487	10/20/2018	340667	194130	1110
v8.1.0100	06/23/2018	335645	192437	1102
v8.1.0000	05/17/2018	334699	191788	1102
v8.0.1500	02/11/2018	332915	192405	1095
v8.0.1300	11/16/2017	331520	191594	1092
v8.0.0700	07/08/2017	326234	192879	1091
v8.0.0200	01/17/2017	318230	191467	1083
v7.4.2000	07/08/2016	316967	189407	1084
v7.4.770	07/10/2015	310204	180695	1056
v7.4.360	07/09/2014	305857	178408	1054
v7.4.001	08/14/2013	302699	178417	1047
v7.3.630	08/15/2012	290891	174178	1003
v7.3.270	08/10/2011	286054	171528	987

Table 4.4: Meta data of the *Vim* selected commits

4.4 Results

The results of our variability-aware *CIA* are depicted in the form of bar charts in Figure 4.1, Figure 4.2, and Figure 4.3 for *OpenVPN*, *BusyBox*, and *Vim* subject systems respectively. Notice, that in each bar chart the x-axis corresponds to different pair-wise commit comparisons and in each comparison, there are five rectangular bars, each of which corresponds to a distinct piece of information as described in the previous section. The y-axis in the bar chart corresponds to the magnitude of features. Furthermore, the comparisons are rendered in the bar chart in an ascending order (by date) i.e. starting from the old commits comparison to the new commits comparison (from left to right).

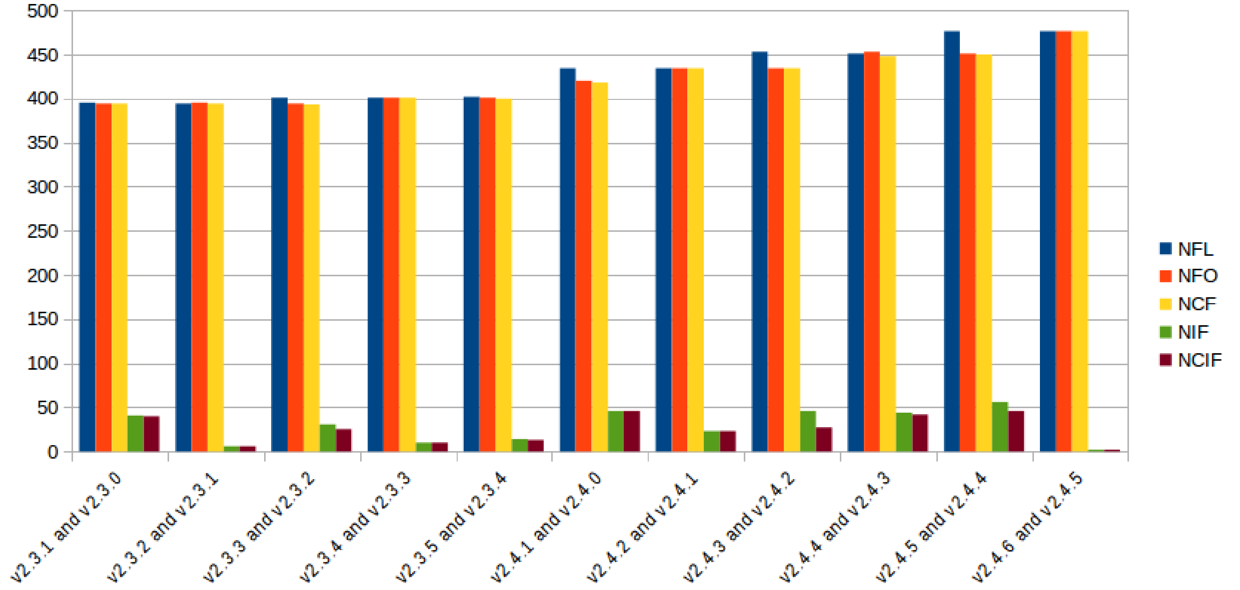


Figure 4.1: *CIA* results for *OpenVPN*.

If we analyze the results for *OpenVPN* and *BusyBox* subject systems (see: Figure 4.1 and Figure 4.2) given in the bar charts, it is apparent that for each pair-wise comparison (between the commits) for which we have performed our variability-aware *CIA*, the magnitude of impacted features (*NIF*) and common impacted features (*NCIF*) is relatively very small compared to the total number of features involved in the comparison (i.e. $NFL \cup NFO$). This implies that the changes do not have many implications on the features involved in the respective systems.

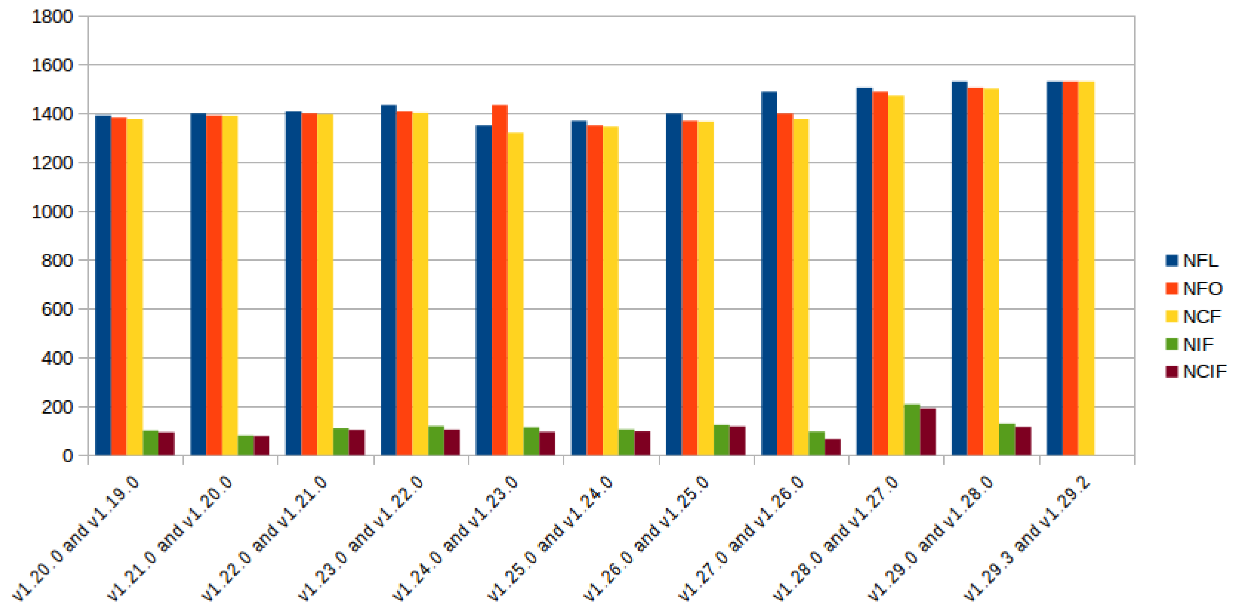


Figure 4.2: *CIA* results for *BusyBox*.

However, when it comes to *Vim* system (see: Figure 4.3) the magnitude of impacted features (*NIF*) and common impacted features (*NCIF*) is relatively much higher

which depicts that changes have more often impacted the features involved in the respective system. Moreover, in each subject system during the latest commit comparison even though the magnitude recorded for *NFL*, *NFO*, and *NCF* is maximum, yet the magnitude recorded for *NIF* and *NCIF* is least. That's because the time span between the commits involved in the latest commit comparison is short compared to the time span in other commit comparisons.

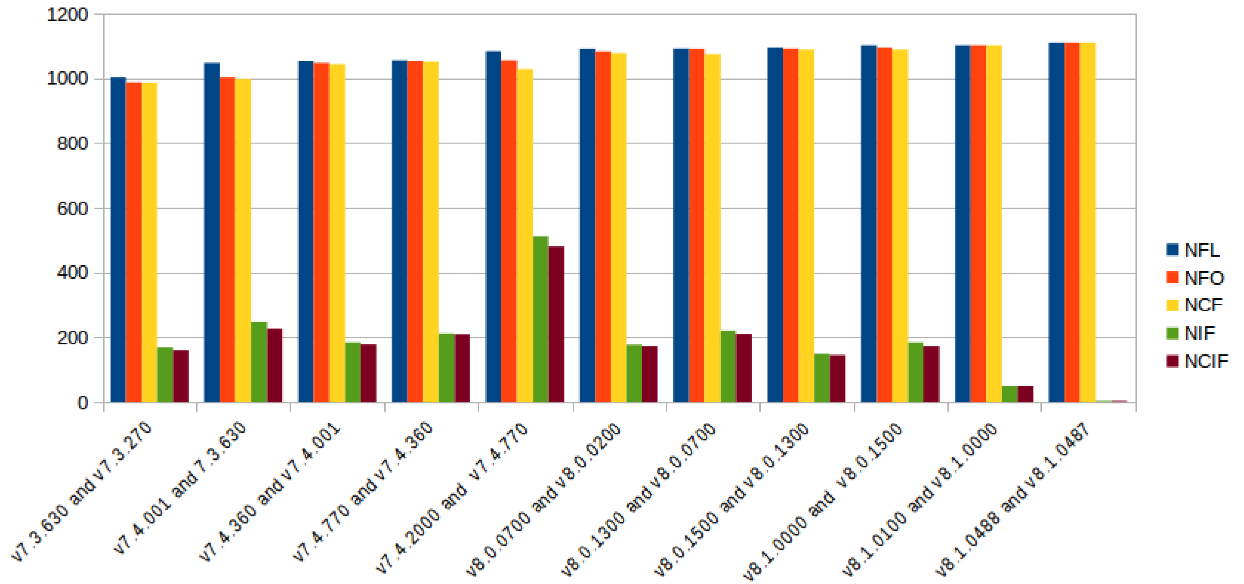


Figure 4.3: *CIA* results for *Vim*.

4.4.1 Arguing RQ's

We evaluated our *CIA* approach against three systems, each of which belongs to distinct application domain and from each system we took several commits and performed *pair-wise comparisons*. Moreover for brevity, while arguing the research questions whenever we use the term comparison it actually refers to the execution of our variability-aware *CIA* between the distinct pair of commits from each subject system to determine the impact of changes in the latest commit with respect to the older commit in each pair. To this end, estimating the set of impacted features (see: Section 4.3).

4.4.1.1 *RQ1*

The final results shown in the bar charts (see: Figure 4.1, Figure 4.2, and Figure 4.3) for each subject system clearly conclude that our variability-aware *CIA* approach very comprehensively determines the set of affected features from each subject system.

	Between	Feat. involved	Feat. Impacted	Percentage
OpenVPN	v2.4.6 & v2.4.5	476	2	0.42%
BusyBox	v1.29.3 & v1.29.2	1531	0	0.00%
Vim	v8.1.0488 & v8.1.0487	1110	3	0.27%

Table 4.5: Features impacted during latest commit comparisons

For the proof of concept, we have manually inspected the most latest commits and computed the set of impacted features for each subject system. That way later when we execute our variability-aware *CIA* between these latest pairs of commits i.e. *v2.4.6 & v2.4.5* from *OpenVPN*, *v1.29.3 & v1.29.2* from *BusyBox*, and *v8.1.0488 & v8.1.0487* from *Vim* - we can verify our results against the beforehand taken readings. Our intuition for selecting only the latest pair of commits from each subject system for the proof of concept is based on the common speculation that the number of changes between the most latest commits (revisions) of a system must be fewer (since the time period is typically short), consequently the number of impacted features will also be fewer too. Hence, this helps us in better inquire whether our variability-aware *CIA* is performing efficiently and providing with accurate results?

Table 4.5, present the data regarding the ratio of impacted features with respect to the total number of features involved after executing our variability-aware *CIA* between the latest commit pairs from each subject system. Note, that the number of features impacted is only `2` out of 476 features with a percentage of `0.42%` in *OpenVPN*, the `2` features impacted are `ENABLE_CRYPTO` and `INTERVAL_DEBUG`, where the *change frequency* (see: Section 4.4.1.3) of `ENABLE_CRYPTO` (*14* times) is recorded greater then `INTERVAL_DEBUG` (only *1* time). Similarly, in *BusyBox* none of the features are impacted by the changes which mean that the ratio of change impact with respect to impacted features is `0%`. Finally, in the case of *Vim* `3` features are impacted out of `1110` features in total, with a ratio of `0.27%`. The feature impacted includes `EXITFREE`, `FEAT_QUICKFIX`, and `PROTO` where the *change frequency* of the impacted features are *2*, *50* and *50* times respectively. Nevertheless, based on the results that we achieved through the execution of our variability-aware *CIA* between the selected pairs of latest commits (from each subject system), we found that it matches 100% with the results, that we gathered through our manual inspection. Thus, verifies the accuracy and reliability of our *CIA*.

4.4.1.2 *RQ2*

It is quite evident from the resultant bar charts (see: Figure 4.1, Figure 4.2, and Figure 4.3) that our proposed variability-aware *CIA* is well-versed in determining the common impacted features in a variable software system. We will interpret this in detail with the help of two line charts. Figure 4.4 depicts the first line chart which shows the ratio of impacted features (*NIF*) with respect to the number of features

involved in the latest commit (*NFL*) for each pair-wise comparison. Then, Figure 4.5 depicts the second line chart which shows the ratio of common impacted features (*NCIF*) with respect to the impacted features (*NIF*) for each pair-wise comparison. Each individual line in the respective line charts corresponds to a particular subject system.

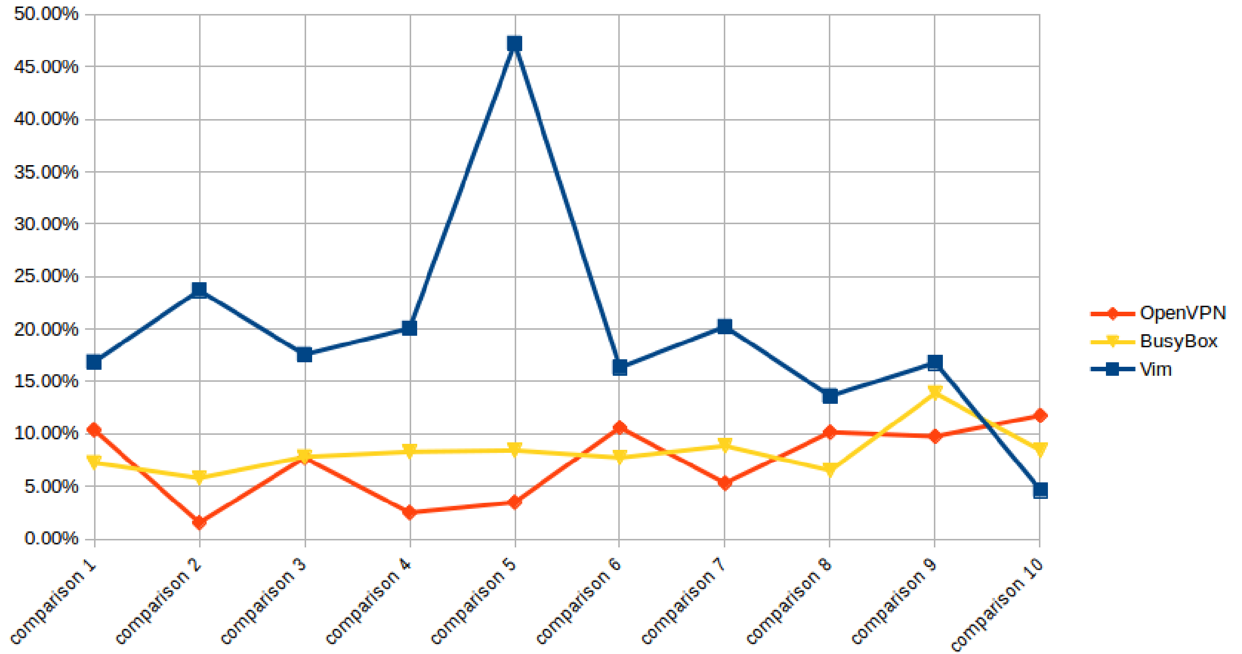


Figure 4.4: Percentage of *NIF* w.r.t *NFL*.

Note in Figure 4.4, that for *OpenVPN* and *BusyBox* subject systems the ratio of impacted features in all the comparisons never recorded above then 15% and in most of the comparisons it remained between 1% to 10%. However, in case of *Vim* system, the ratio of impacted features in most of the comparisons remained between 15% to 25% except in comparison 5 - where it shoots substantially to almost 50%, and, in comparisons 8 and 10 where it drops to 14% and 4% respectively. This indicates that in *Vim* more features are affected by the changes compared to the other two systems, which is probably because of the higher number of features involved in the *vim* systems. Now, if we analyze Figure 4.5 we can clearly observe that our variability-aware *CIA* has competently computed the set of all the common impacted features (*NCIF*) for each subject system with respect to the impacted features (*NIF* - computed in Figure 4.4) for each comparison.

Note, that the ratio of common impacted features (*NCIF*) in most of the comparisons remained between 80% to 100% - which indicates that the features impacted by the changes are most of the time exists in both the commits that we are comparing from a particular subject system. However, during comparison 8 for both *OpenVPN* and *BusyBox* subject systems, we see that the ratio of common impacted features drops somewhat to 70% and 60% respectively.

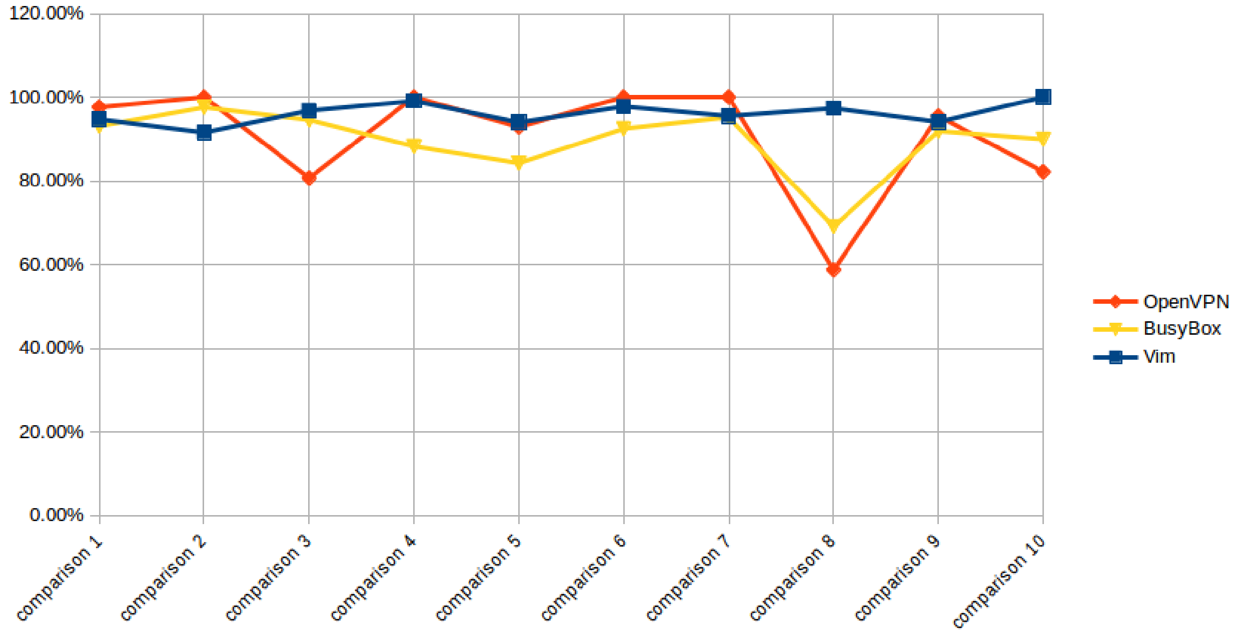


Figure 4.5: Percentage of *NCIF* w.r.t *NIF*.

4.4.1.3 *RQ3*

We performed eleven comparisons in total between different commits (releases) from each subject system as apparent in the resultant bar charts (see: Figure 4.1, Figure 4.2, and Figure 4.3). Besides, we can also witness from the results that there are impacted features involved in each comparison. Additionally, our variability-aware CIA also facilitates the end-user with the stats regarding the most frequently impacted features in each subject system. Table 4.6 reveal the list of most frequently impacted features in each subject system, where the column `NFIC` (feature involved in comparison) corresponds to the number of times a feature constant appeared in distinct comparisons collectively, and, the column `frequency` corresponds to the number of time a feature constant is impacted by a change in respective comparisons.

Note, that in *OpenVPN* the most frequently impacted feature is `ENABLE_CRYPTO` which is affected by the changes in all the eleven comparisons, also, the feature is impacted very often (i.e. `911` times). Similarly, in *BusyBox* system, `ANDROID`, `__ANDROID__`, and `ENABLE_FEATURE_EDITING` are the most frequently impacted features which were affected ten out of eleven times in the performed comparisons. However, it is also apparent from the data that even-though `ANDROID`, `__ANDROID__` are affected in all the comparison, yet, the frequencies of getting affected by the changes of these features (i.e. just `15`, `15` times each) are relatively very small compared to `ENABLE_FEATURE_EDITING` feature (i.e. `908` times). Finally, in *vim* system, `PROTO` and `FEAT_QUICKFIX` are the most frequently impacted features and are recorded as affected in all eleven out of eleven comparisons. But again `PROTO` feature is impacted very intensively (i.e. `62398` times) compared to `FEAT_QUICKFIX` feature (i.e. `5699` times) during the respective comparisons.

	feature	NFIC	frequency
OpenVPN	ENABLE_CRYPTO	11	911
BusyBox	ANDROID	10	15
	__ANDROID__	10	15
	ENABLE_FEATURE_EDITING	10	908
Vim	PROTO	11	62398
	FEAT_QUICKFIX	11	5699

Table 4.6: Frequently impacted features

4.4.2 Summing-up

It is pretty obvious from the results that our variability-ware *CIA* is extremely reliable and is not only limited to evaluating the impacted statements in the source code but is also quite effective in determining the impacted features (their change frequency and so on). In particular, we performed our evaluation on three open-source systems i.e. *OpenVPN*, *BusyBox*, and *Vim* with `470`, `1537`, and `8975` features involved respectively. For all the comparisons that we have performed between distinct commits from each subject system, the overall proportions of the impacted features in *OpenVPN*, *BusyBox*, and *Vim* is recorded up to `7.47%`, `8.33%`, and `19.67%` respectively - which concludes that the impact of changes between different revisions (commits) of these systems have never been too immense with respect to the underlying variable source code.

5. Related Work

In this chapter, we discuss other significant works that are more similar and related to our research work. In particular, we highlight works that are submitted by other authors and compare them with our proposed approach.

During the evolution of a software system, the changes are inevitable and hence there has been a vital requirement for some explicit means to assess and evaluate the impact of a change on existing concepts and artifacts. Thus, several works have been performed since the last twenty years in this research area and the number of studies published in the area of change impact analysis is enormous [27]. In particular, we structure our comparison to related work into work concerning.

- *Program slicing.*
- *Variability-aware data-flow analysis.*
- *Change impact analysis.*

Program slicing

In the context of program slicing *Livadas et al.* has presented a technique for mapping nodes of dependence graphs to exact locations in un-preprocessed source code [33]. *Vidács et al.* on the other hand proposed slicing of *CPP* macro expansions as an extension to slicing of *C++* programs [44]. However, both these approaches are basically meant for analyzing macros in a program instead of the variability caused by conditional compilation. Furthermore, these approaches are only limited to pre-processed code, unlike our approach where the program slicing is performed on the un-preprocessed code.

Variability-aware data-flow analysis

In the context of variability-aware data-flow analysis, *Liebig et al.* proposed a static analysis of *C* based programs in the existence of variability, especially control-flow and liveness analyses. They also proved the ease of scalability of their approach [32]. *Kästner et al.* introduce the tool *TypeChef*, which parses un-preprocessed *C* source code and encodes the variability in the form of abstract syntax tree utilizing presence conditions [23]. *Brabrand et al.* also proposed different ways for uplifting support for standard intraprocedural data flow analysis in product lines via feature-sensitive analyses which eventually assist in analyzing the entire program space of a preprocessor-based product line [9]. *Frederik et al.* proposed a technique for precise slicing of un-preprocessed *C* programs and as a result, provides with an extended PDG concept which directly represents all possible variants of a program. They used *TypeChef* as an underlying infrastructure for conducting the data-flow analysis [21]. In our approach for data-flow analysis, we extended the work by *Sven Apel et al.* [30]. They present the tool *cppstats*, which we used as the underlying infrastructure for our analyses.

Change impact analysis

In the context of change impact analysis, *Florian et al.* has presented a technique which is based on program slicing techniques and a conditional system dependence graph (CSDG) - an extension of a system dependence graph (SDG). The approach can deal with load-time configuration options i.e. variability information load from a file or provided through program arguments [2]. In contrast to that our approach is mainly a variability-aware program analysis techniques which use program slicing technique (forward slicing) and a variable control flow graph (CFG). Moreover, our approach extract variability information from the source code i.e. *CPP* annotations.

6. Conclusion and Future Work

The analysis of a highly-configurable variable software system has always been a challenging task because of inherent complexity, induced by variability. Thus, different variability-aware program analysis techniques have been introduced for analyzing the space and scope of program variants. In this thesis, we have presented a variability-aware change impact analysis approach, in which we start with variability-aware analysis to extract all the variability information in the system, then, we perform program slicing (forward slicing) for the exploration, analysis, and manipulation of source code. Finally, we perform our core *CIA* analysis for estimating the affected features in the system which eventually lead us to identify all the impacted variants in a software system. For variability-aware analysis, we used *cppstats* framework and for slicing, we incorporated *srcML* slicing utility called *srcSlice*. Then, we evaluated our approach against three popular open-source systems, to investigate the reliability and performance of our *CIA* technique.

Regarding *RQ1* we found high accuracy rate of our variability-aware *CIA* approach which benefits developers and maintainer of a system with more reliable change impact analysis. With respect to *RQ2*, determining the set of common impacted features between different commits (revisions) make ease in tracking the features which are changing over time. Regarding *RQ3*, the estimation of more frequently impacted features allows the use of our technique in typical maintenance and development tasks. Our goal is to assist programmers and maintainers of variable systems in general tasks, such as testing or maintenance, by giving useful information in an expedited way regarding rippling effects transpired due to changes, which ultimately increase both, efficiency as well as scalability of a software system.

Although, we laid the foundations for a valuable and scalable program change impact analysis technique for variable software systems. Yet, there are some limitations regarding practical usage which we would like to address as future works:

- **The limitation to a single language:**

  Currently, our proposed technique only facilitates the systems which are programmed in *C* language and are using CPP conditional compilation directives (`#ifdef`) as a medium for representing variability in the source code. However, as a future work, we would like to extend our system so that it can also adapt and facilitates other the systems as well, which are written in programming languages other than *C* language but are using CPP annotations e.g. *C++*.

- **Increase efficiency and effectiveness:**

  The final reports generated through our variability-aware `CIA` also stocks the expressions which involve the feature constants (see: Section 3.2.5.2). We, therefore, intend to explore this information to identify the affected features and consequently the products which involve those features. To this end, the overhead of testing the entire set of product variants will reduce to only those product variants which actually undergoes through changes. Hence, increases the efficiency and effectiveness of our approach and outperform various sampling approaches as well.

- **Analyzing change interaction:**

  Also using the information we retrieved from our variability-aware *CIA* e.g. expressions involving feature constants, start and end line numbers (see: Section 3.2.5.2) we intend to further explore the possible implicit dependencies between the features via exploiting the change impact patterns.

Bibliography

[1] Hakam W Alomari, Michael L Collard, Jonathan I Maletic, Nouh Alhindawi, and Omar Meqdadi. srcslice: very efficient and scalable forward static slicing. *Journal of Software: Evolution and Process*, 26(11):931–961, 2014. (cited on Page 18)

[2] Florian Angerer, Andreas Grimmer, Herbert Prahofer, and Paul Grunbacher. Configuration-aware change impact analysis (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 385–395. IEEE, 2015. (cited on Page 2 and 46)

[3] S Apel, D Batory, C Kästner, and G Saake. Feature-oriented software product lines: Concepts and implementation, berlin/heidelberg, 2013, 308 pages. Technical report, ISBN 978-3-642-37520-0. URL http://www. springer. com/computer/swe/book/978-3-642-37520-0, 2013. (cited on Page 1, 5, 6, 8, and 10)

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines.* Springer, 2016. (cited on Page 5)

[5] Sven Apel, Sergiy Kolesnikov, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich. Access control in feature-oriented programming. *Science of Computer Programming*, 77(3):174–187, 2012. (cited on Page 8)

[6] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 432–441, New York, NY, USA, 2005. ACM. (cited on Page 11)

[7] Robert S Arnold. *Software change impact analysis.* IEEE Computer Society Press, 1996. (cited on Page 11)

[8] Shawn A Bohner et al. Impact analysis in the software change process: a year 2000 perspective. In *icsm*, volume 96, pages 42–51, 1996. (cited on Page 1)

[9] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. In *Transactions on Aspect-Oriented Software Development X*, pages 73–108. Springer, 2013. (cited on Page 46)

[10] Michael L Collard. Addressing source code using srcml. In *IEEE International Workshop on Program Comprehension Working Session: Textual Views of Source Code to Support Comprehension (IWPC'05)*. Citeseer, 2005. (cited on Page 16)

[11] Michael L Collard, Michael J Decker, and Jonathan I Maletic. Lightweight transformation and fact extraction with the srcml toolkit. In *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pages 173–184. IEEE, 2011. (cited on Page 15, 16, and 17)

[12] Michael L Collard, Michael John Decker, and Jonathan I Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 516–519. IEEE, 2013. (cited on Page 16)

[13] Tzilla Elrad, Robert E Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001. (cited on Page 8)

[14] Markus Feilner. *OpenVPN: Building and integrating virtual private networks.* Packt Publishing Ltd, 2006. (cited on Page 34)

[15] Tie Feng and Jonathan I Maletic. Applying dynamic change impact analysis in component-based architecture design. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2006. SNPD 2006. Seventh ACIS International Conference on*, pages 43–48. IEEE, 2006. (cited on Page 11)

[16] Gabriel Coutinho Sousa Ferreira, Felipe Nunes Gaia, Eduardo Figueiredo, and Marcelo de Almeida Maia. On the use of feature-oriented programming for evolving software product lines—a comparative study. *Science of Computer Programming*, 93:65–85, 2014. (cited on Page 7 and 8)

[17] Brady J Garvin and Myra B Cohen. Feature interaction faults revisited: An exploratory study. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 90–99. IEEE, 2011. (cited on Page 10)

[18] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering*, 21(2):449–482, 2016. (cited on Page 20)

[19] Mohammad-Amin Jashki, Reza Zafarani, and Ebrahim Bagheri. Towards a more efficient static software change impact analysis method. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '08, pages 84–90, New York, NY, USA, 2008. ACM. (cited on Page 11)

[20] Kyo C Kang, Jaejoon Lee, and Patrick Donohoe. Feature-oriented product line engineering. *IEEE software*, 19(4):58–65, 2002. (cited on Page 6)

[21] Frederik Kanning and Sandro Schulze. Program slicing in the presence of preprocessor variability. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 501–505. IEEE, 2014. (cited on Page 46)

[22] Christian Kästner and Sven Apel. Integrating compositional and annotative approaches for product line engineering. In *Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, pages 35–40, 2008. (cited on Page 7)

[23] 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 *ACM SIGPLAN Notices*, volume 46, pages 805–824. ACM, 2011. (cited on Page 8 and 46)

[24] 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 *Proceedings of the 31st International Conference on Software Engineering*, pages 611–614. IEEE Computer Society, 2009. (cited on Page 7)

[25] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. Typechef: toward type checking# ifdef variability in c. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, pages 25–32. ACM, 2010. (cited on Page 10)

[26] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997. (cited on Page 8)

[27] Steffen Lehnert. A review of software change impact analysis. 2011. (cited on Page 45)

[28] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, 2013. (cited on Page 1 and 11)

[29] Jörg Liebig. *Analysis and Transformation of Configurable Systems.* PhD thesis, Citeseer, 2015. (cited on Page 20)

[30] 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*, pages 105–114. ACM, 2010. (cited on Page 10, 20, and 46)

[31] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 191–202. ACM, 2011. (cited on Page 9 and 20)

[32] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 81–91. ACM, 2013. (cited on Page 1, 10, and 46)

[33] Panos E Livadas and David T Small. Understanding code containing preprocessor constructs. In *Program Comprehension, 1994. Proceedings., IEEE Third Workshop on*, pages 89–97. IEEE, 1994. (cited on Page 45)

[34] Jihen Maâzoun, Nadia Bouassida, and Hanêne Ben-Abdallah. Change impact analysis for software product lines. *Journal of King Saud University-Computer and Information Sciences*, 28(4):364–380, 2016. (cited on Page 7 and 11)

[35] Jonathan I Maletic, Michael L Collard, and Andrian Marcus. Source code files as structured documents. In *Program comprehension, 2002. proceedings. 10th international workshop on*, pages 289–292. IEEE, 2002. (cited on Page 16)

[36] Christian D Newman, Tessandra Sage, Michael L Collard, Hakam W Alomari, and Jonathan I Maletic. srcslice: a tool for efficient static forward slicing. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pages 621–624. IEEE, 2016. (cited on Page 18)

[37] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11, 2011. (cited on Page 10)

[38] Steven Oualline. *Vi iMproved*. New Riders Publishing, 2001. (cited on Page 35)

[39] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *International Conference on Software Product Lines*, pages 77–91. Springer, 2010. (cited on Page 8)

[40] Periklis Sochos, Ilka Philippow, and Matthias Riebisch. Feature-oriented development of software product lines: mapping feature models to the architecture. In *Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*, pages 138–152. Springer, 2004. (cited on Page 5)

[41] Richard M Stallman and Zachary Weinberg. The c preprocessor. *Free Software Foundation*, 1987. (cited on Page 8)

[42] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software: Practice and experience*, 35(8):705–754, 2005. (cited on Page 6)

[43] Frank Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica, 1994. (cited on Page 12)

[44] László Vidács, Árpád Beszédes, and Tibor Gyimóthy. Combining preprocessor slicing with c/c++ language slicing. *Science of Computer Programming*, 74(7):399–413, 2009. (cited on Page 45)

[45] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981. (cited on Page 11 and 13)

[46] Nicholas Wells. Busybox: A swiss army knife for linux. *Linux Journal*, 2000(78es):10, 2000. (cited on Page 34)