JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
**Dipl.-Ing.
Florian Angerer**

Submitted at
**Institute for
System Software**

Supervisor and
First Examiner
**a.Univ.-Prof.
Dipl.-Ing. Dr.
Herbert Prähofer**

Second Examiner
**Prof. Dr.-Ing.
Sven Apel**

January 2017

Configuration-aware Program Analysis for Maintenance and Evolution in Industrial Software Product Lines

Doctoral Thesis
to obtain the academic degree of
Doktor der technischen Wissenschaften
in the Doctoral Program
Technische Wissenschaften

**JOHANNES KEPLER
UNIVERSITY LINZ**
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

Due to increased market demands for highly customized solutions, software systems are often organized as software product lines (SPLs) where individual solutions are created from a common code base. SPLs support managing common functionality relevant for most customers while still allowing to provide specific extensions for particular needs. Software development in SPLs, however, results in new challenges regarding software development processes and tool support. Specifically, the many possible variants and versions calling for effective variability management. Supporting this variability in the software, however, leads to increased design complexity and maintenance effort.

This thesis presents static program analysis methods and tool support considering the variability and the configuration in SPLs. Based on a representation of program dependencies in the form of a conditional system dependence graph (CSDG), which also encodes the configuration options of a SPL, it introduces methods supporting different maintenance and evolution scenarios in the SPL context. The inactive code detection method allows to automatically identify code in a product, which is inactive due to the specific product configuration. Further, a method for interprocedural and configuration-aware change impact analysis (CA-CIA) allows determining possibly impacted products when changing source code of a SPL and also supports engineers who are adapting specific product variants after an initial configuration.

The methods presented in this thesis particularly focus on the delayed variability principle that allows performing configuration-aware program analysis for software systems using load-time configuration options. So far, research has focused on compile-time variability that allows to generate software products by including or excluding program code. However, load-time variability also imposes new challenges for developing, testing, and analyzing variable software systems. Therefore, a comparative experiment has been conducted to compare the well-established lifted analysis approach to the delayed variability approach. Results show that the delayed variability approach is significantly faster but in some cases sacrifices precision for performance.

Kurzfassung

Um die hohen Anforderungen des Software Markts erfüllen zu können, werden immer häufiger Software Produktlinien (SPLs) eingesetzt, welche es erlauben individuelle Lösungen aus vorhandenen Bausteinen zu erstellen. Dabei verwaltet eine SPL Basisfunktionen die für einen Großteil der Kunden benötigt werden wobei eben noch spezielle Lösungen ergänzt werden können. Die Vorteile von SPLs wurden vielfach gezeigt, aber durch diese vergleichsweise junge Art von Softwareorganisation entstehen auch neue Herausforderungen, die den Entwicklungsprozess und die Entwicklungswerkzeuge betreffen. Um genau zu sein, die Unzahl an möglichen Produktvarianten bringt einen höheren Aufwand bei Entwurf und Wartung mit sich und Entwickler benötigen Unterstützung bei diesen Aufgaben.

Diese Arbeit stellt daher statische Programmanalysemethoden und Werkzeuge vor, welche Variabilität bzw. Konfiguration in Software Produktlinien unterstützen. Dafür wird eine Datenstruktur namens Conditional System Dependence Graph (CSDG) verwendet, welcher alle Daten- und Kontrollabhängigkeiten im Programm repräsentiert und auch die Variabilität abbildet. Die erste Methode verwendet den CSDG um inaktiven Source Code von konkreten Produktvarianten automatisch zu erkennen. Der zweite Ansatz erlaubt es eine globale Auswirkungsanalyse für Änderungen durchzuführen, wobei die Variabilität des Programms berücksichtigt wird. Dieser Ansatz ist im speziellen dafür gedacht, Entwickler bei der Wartung der Produktlinie und bei der Adaptierung von Varianten zu unterstützen.

Die vorgestellten Methoden verwenden das *Delayed Variability* Analyseprinzip um Programmanalyse unter der Berücksichtigung von Konfiguration durchzuführen, wobei speziell auf Konfigurationsoptionen die beim Programmstart geladen werden, abgezielt wird. Das ist auch der wesentliche Unterschied zu bestehenden Ansätzen, die als *Lifted Analysis* bezeichnet werden und sich meist auf Konfiguration während der Übersetzungszeit konzentrieren. Da es aber Überschneidungen bei den Ansätzen gibt, präsentiert diese Arbeit auch einen Vergleich zwischen den Ansätzen für Konfiguration zum Programmstart und zur Übersetzungszeit. Die Ergebnisse zeigen einen deutlichen Geschwindigkeitsvorteil für Delayed Variability, jedoch zum Preis von Genauigkeit.

Acknowledgments

First of all, I would like to thank my advisor Herbert Prähofer for guiding and supporting my work over several years. Your efforts enabled me to do a Ph.D. study and you were always convinced of my work's potential. Also many thanks to Paul Grünbacher, who supported me and my research in word and deed.

Further, I want to thank my colleagues and friends. In particular, many thanks to my fellow teammate Daniela Rabiser for the excellent research cooperation and direct support for many issues during our joint time in the research laboratory. Special thanks goes to my former colleagues and today's friends Peter Hofer and Andreas Grimmer for direct and indirect support and encouraging words. I learned a lot from you and we really had a good time. Also many thanks to my colleagues Peter Feichtinger, Mario Kromoser, and Daniel Hinterreiter for their valuable contributions.

Furthermore, I like to thank our industry partner KEBA AG, in particular, Ernst Steller, Dietmar Berlesreiter, Michael Petruzelka, and Gottfried Schmidleitner which continuously provided input and feedback for my work. Additionally, I want to thank Lukas Linsbauer (JKU), Sven Apel (University of Passau), Andrzej Wąsowski (ITU Copenhagen), and Claus Brabrand (ITU Copenhagen) for their research cooperation. I also want to thank the students David Auinger and Alois Mühleder who contributed to my work.

Last but not least I also thank my family and in particular my wife Stefanie for always encouraging me to continue and finally finish my Ph.D study.

Contents

Chapter 1

Introduction

Many software companies target to produce software for a mass-market to increase their return on investment, while on the other hand, customers increasingly demand individual solutions [Clements02]. Therefore, systems are often organized as software product lines (SPLs), which allow managing the common functionality relevant for most customers and the variability. In such a context, customer-specific solutions are often built in a multi-stage manner: system variants are first derived from the SPLs and then adapted and extended to meet the specific requirements of customers [Kästner08].

Dealing with variability is essential in SPLs and techniques are needed for composing and customizing systems. Many existing approaches dealing with variability assume that the source code is annotated directly with variability information, which is the case, e.g., in annotation-based product lines that use preprocessors [Kästner11]. However, other variability mechanisms, such as load-time configuration options provided via a file or via program arguments, play an equally important role [Lillack14].

Such variability mechanisms, however, increase the challenges for developing, maintaining, testing, and analyzing systems and tool support for developers becomes essential. For example, when features are changed in the original SPL (e.g., to fix a bug) derived product variants often need to be updated.

In addition, complex software systems commonly consist of multiple interdependent programs written in different languages. For example in the domain of industrial automation, a software system usually consists of a real-time part typically controlling time-critical processes and a non-real-time part comprising graphical user interfaces.

Current program analysis approaches and tools have limitations to support variability and multiple programming languages. However, research on variable software systems has progressed significantly. For instance, researchers in software product lines and feature-oriented software development have developed family approaches that allow analyzing the whole space of software variants by exploiting commonalities between variants [Thüm14]. However, these techniques are not designed to support variable software using load-time configuration and to be applied in multi-language software systems. This means that, although existing approaches can deal with variability, most

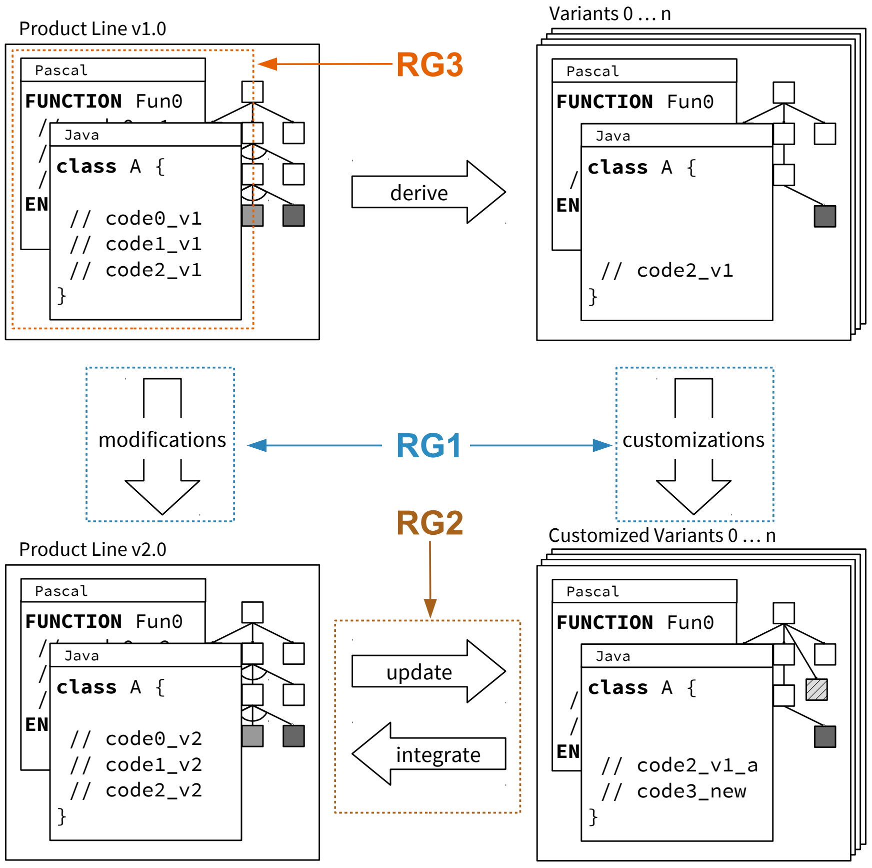


Figure 1.1: Customization and evolution process of a multi-language variable software system.

assume compile-time bound variability, e.g., preprocessor directives. Furthermore, analysis often ends at language borders.

This thesis therefore aims at developing static program analysis for multi-language variable software systems using load-time configuration options. Specifically, the work focuses on configuration-aware program analysis supporting developers during maintenance and evolution.

In the following, we present the research goals and the general approach for achieving them. Next, we explain the research context, i.e., the cooperation with an industry partner, which further motivates the research goals. Then, we outline the research contributions of this thesis and explain its overall structure.

1.1 Research Goals and Approach

Figure 1.1 illustrates a product derivation and evolution scenario common in SPL development processes. First, a product variant is derived from the SPL platform by selecting features needed by a customer. Developers then adapt and extend the derived variants according to the customer's requirements resulting in customized variants. The customized variants often implement new or improved functionality, which is then frequently integrated back into the SPL. Similarly, when changes are made in the SPL (e.g., to fix a bug), the customized variants often need to be updated. Assessing the impact of changes, reducing the complexity of source code, and providing multi-language capabilities is essential in such a context and developers need to have appropriate tools and techniques. The research goals derived from this scenario are therefore:

RG1 Support for assessing the impact of changes in variable software. Modifying the source code of the product family or of partially configured product variants is a frequent task and therefore developers need support for determining the possible impact of changes.

RG2 Methods for creating views with reduced source code complexity during maintenance. This is necessary because the product family or partially configured product variants often contain lots of source code irrelevant for a specific maintenance task.

RG3 Program analysis support for multi-language software systems is important, because large-scale software systems usually consist of subsystems written in different programming languages. The subsystems are interdependent and changes in one part may affect other parts.

We rely on static program analysis methods for achieving these research goals. In general, the research goals call for variability-aware and cross-language analysis support. To address the goals for the analysis of highly-configurable software systems and the missing capabilities of existing approaches for specifically handling values of configuration options, we propose the concept of *delayed variability analysis*. The primary requirements for delayed variability analysis are (i) the reuse of standard implementations of data flow analyses, and (ii) the propagation of the configuration values in the program for providing variability information. The delayed variability analysis works by performing an analysis completely variability-oblivious, i.e., ignoring any variability in the first place, and then recovering the variability and augmenting the results. This has the advantage that just one analysis is performed. This strategy avoids the problem of lots of explicit intermediate analysis results because the analysis is done just once regardless of variability. However, static program analysis always has to make a tradeoff between run-time performance and precision. The drawback of this strategy is the loss in precision because the delayed variability analysis needs to overapproximate situations to be sound.

These requirements are achieved by building a graph structure using standard control flow and data flow analysis implementations, extracting variability information from the program, and adding this information to the graph structure. The actual analysis is then performed on the graph structure considering the available variability information.

1.2 Research Context

The research was conducted in collaboration with KEBA AG, a company developing hardware and software platforms and solutions for industrial automation. The development of automation solutions is a multi-stage process involving different stakeholders: KEBA develops and produces hardware and software platforms with associated tool support. The hardware and software solutions then enable the customers of KEBA, usually OEMs of manufacturing machines, to develop their own products and customized automation solutions. Thus, the software solutions of KEBA have to be highly configurable to address the manifold requirements of different market segments and customers. Figure 1.2 shows the typical development process of the industry partner KEBA which is organized as a multi-stage process relying on several platforms. The *Automation Platform* on top of the figure is the central pillar providing runtime systems and basic capabilities for automation solutions. It is a software product line which allows instantiating different variants. Based on the automation platform, KEBA provides several domain-specific solutions in the form of configurable and adaptable product lines. An example is the KePlast platform [Lettner13] providing comprehensive capabilities for the automation of injection molding machines. KePlast implements the following subsystems: a configurable control core implemented in a proprietary dialect of the IEC 61131-3 standard, a visualization system written in Java, and programming and configuration tools to customize the solution based on existing variants. The platform is then used as a basis for concrete customer-specific products. The customer products are created in a staged configuration and adaptation process: First, the configuration tool *Application Composer (AppCo)* [Lettner13] is used to derive a basic solution. Then, this initial product is further configured, adapted, and extended. For instance, engineers may add new features or modify existing ones to meet the customer requirements at hand. In this process, engineers usually also implement solutions which have potential to be reused and, therefore, should become part of the core platform.

In an exploratory case study, we investigated KEBA's software systems and development processes to derive characteristics and evolution challenges (cf. [Lettner14a, Lettner14b]). First, in workshops with managers and project leaders we got an initial understanding of development processes and perceived problems. Then, by archival analysis we investigated their solution artifacts, in particular, how variability is realized and how engineers perform derivation and adaptions. Finally, we conducted an interview study with application developers to find out about their lived working processes

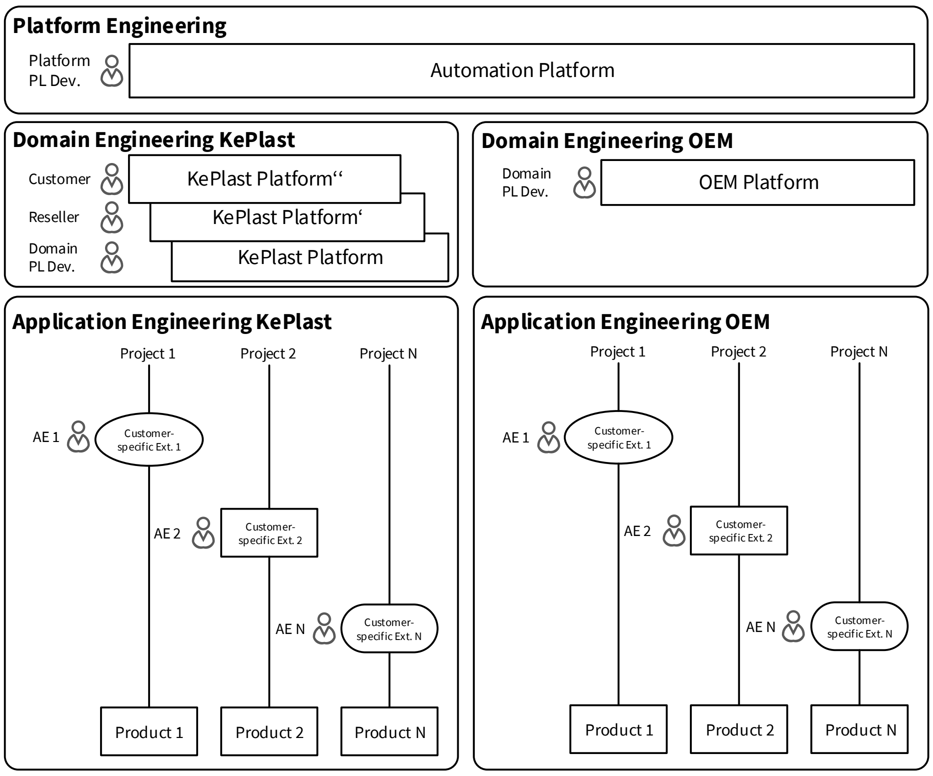


Figure 1.2: A brief overview of KEBA’s development process [Lettner14a]. The software system is organized in a layered structure and comprises several platforms. The common technological platform is the Automation Platform and several domain-specific solutions are built by KEBA as well as external companies (OEMs) at the top of the Automation Platform. The domain-specific solutions are further adapted for the final customers.

and perceived difficulties. In total, ten developers employed by KEBA and two further developers of a customer of KEBA, a customer is an OEM developing solutions based on KEBA’s automation platform, were interviewed.

Based on the exploratory case study, eight software evolution challenges were derived as listed in Table 1.1. The challenges highlighted with boldface font are particularly relevant for this thesis and its research goals.

In the following paragraphs we describe these challenges in more detail. Please refer to [Lettner14a] for in-depth descriptions of the other challenges, which have been addressed in the PhD thesis by Daniela Rabiser [Rabiser16a].

1. Merging in multi-platform / multi-product environments Merging existing applications and product variants with new platform releases is a major effort for KEBA’s developers. When building customer-specific applications, application engineers (AEs) derive a plain application from a platform and then make changes to meet a customer’s requirements. Upgrading these specific product variants to new platform releases is highly challenging.

	Challenge	Key Findings
1	**Merging in multi-platform / multi-product environments**	Merging requires consideration of customer-specific extensions, pitfalls of older platform versions, and variant-specific bug fixes
2	**Impact analyses across multiple platforms and variants**	Lack of adequate approach to assess change impact on multiple systems and diverse variants
3	Asynchronous release management in multi-platform SECOs	Interdepending software platforms evolved by multiple teams in multiple organizations adhering to different release cycles
4	**Lack of guidance for modifying software**	Huge amount of information on different systems and variants hinders emphasis on relevant pieces of information
5	Lack of systematic reuse of feature implementations	Solutions perfectly suited for reuse are often not discovered
6	**Feature implementations in multiple languages**	Systems exploiting multiple programming languages and technologies challenge feature and platform evolution
7	Aligning product management and developer views on software features	Multiple role-specific perspectives regarding the features of a platform may differ considerably
8	SECO-driven platform evolution	Missing feedback loop supporting communication and coordination between distributed development sites

Table 1.1: Software evolution challenges [Lettner14a].

AEs have only basic support for identifying the changes that were made since the last release. They lack support for determining which software components are relevant for their application and how they can migrate customer-specific extensions to a new platform release. Due to these difficulties only selected new features are migrated. Further, it is common practice in this domain that customers skip several platform releases. This makes merging and upgrading of existing applications even worse.

2. Impact analyses across multiple platforms and variants The developers reported difficulties in predicting the impact of changes on other platforms and product variants when implementing a customer requirement or when fixing a bug. Tool support is currently typically limited to basic tools in IDEs such as simple text search. This leads to high implementation efforts especially if multiple systems and diverse variants need to be considered. Besides, KEBA's developers have to consider multiple programming languages when analyzing the impact of a change, since most changes affect code written in different programming languages.

4. Lack of guidance for modifying software KEBA's software solution covers a wide range of different systems and variants. This makes it difficult for developers following a clone-and-own development approach [Rubin13] to locate the relevant artifacts (e.g., sections in source code) when making changes. For instance, AEs lack mechanisms to reduce complexity by hiding information not needed for a task.

6. Feature implementations in multiple languages KEBA's software systems are implemented in different programming languages including C, C++, Java, a dialect of the IEC 61131-3 standard, and C#. Providing comprehensive tool support is hard given this diversity and even basic development tasks for which good support is available in single-language environments can become challenging. An example is refactoring which becomes complex and time-consuming due to a lack of inter-system and inter-language support. Furthermore, platforms use different mechanisms to deal with configuration settings. It is thus hard to evolve features and platforms as both code and configuration files need to be considered when making changes.

Moreover, an analysis of the KePlast code base showed that the product line platform's variability is mainly implemented using load-time configuration options, i.e., `IF` statements that evaluate configuration settings. Thus, developers currently have to consider the entire code base of the product line when making customer-specific adaptations. During the interviews, participants reported the need to reduce complexity by hiding information not needed for adapting a system, e.g., filtering program elements relevant for a particular configuration.

1.3 Research Contributions

Challenge 1, 2, 4, and 6 from Table 1.1 (cf. Section 1.2) describe issues during development and maintenance of source code and are targeted by this thesis. For example, Challenge 2 *Impact analysis across multiple platforms and variants* calls for a systematic approach for determining the impact of changes considering variability. In the following, we give an overview of the contributions of this thesis and briefly discuss how they address the above challenges. The contributions are then presented in detail in the subsequent chapters of this thesis.

Contribution 1 – Conditional System Dependence Graph

The *conditional system dependence graph (CSDG)* in Chapter 3 is the basic data structure for the configuration-aware program analysis methods of this thesis. The CSDG is based on the system dependence graph (SDG) that has been introduced by Horwitz et al. [Horwitz90] for representing control and data dependencies globally in a program. A SDG is a directed graph representing different kinds of dependencies between program elements. It usually represents control flow and data flow dependencies, but other types of dependency are possible, e.g., definition-use dependencies [Horwitz90].

The CSDG extends the SDG by introducing presence conditions for representing the variability of a program. Mechanisms for implementing variability in the source code are extracted and abstracted for encoding presence conditions for control flow and data flow dependencies. The presence conditions are attached to edges in the CSDG, meaning that the dependency is only enabled if the presence conditions evaluates to true for a certain program configuration. In this way presence conditions are used for representing the variability information of configurable systems.

Our tool chain builds the CSDG by first building an abstract syntax tree (AST) from source code, then performing a data flow and pointer analysis using the Soot program analysis framework [Lam11], followed by extracting the presence conditions from source code, and finally annotating edges in the SDG for building the CSDG. Further, the approach allows integrating different source languages, e.g., the approach allows building a CSDG spanning the industry partner's IEC 61131-3 language and Java. In evaluation studies based on our industry partner's software KePlast we show that building the CSDG is run-time efficient and also scales for large-scale industrial systems.

Contribution 2 – Identifying Inactive Code

Application engineers frequently create customer-specific products in two stages: the required software components are first selected to create an initial product which is then evolved by refining the selected features and adapting the code to meet the customers' requirements in a clone-and-own manner. For instance, developers frequently set configuration options in the code to adjust the product.

However, given that such changes are often necessary in the entire code base, it is hard to know which part of the code is still relevant for the chosen configuration options. This means that engineers need to understand and maintain a lot of code that is potentially inactive in a particular product variant.

The *inactive code detection (ICD)* approach presented in Chapter 4 is a method for reducing the source code complexity by determining code that will never be executed because of the specific product configuration. First, it determines the code which is conditionally executed based on the product configuration at hand. Then, it uses the CSDG to follow control dependencies and, in this way, determines the statements that will never be executed. In a development tool, the code can then be hidden to reduce the code complexity for developers. The evaluation based on industrial systems shows that the approach is run-time efficient as it allows handling large-scale system, it is effective as inactive code size is significant, and it is accurate in the sense that it compares to analysis results from experts.

Further, the ICD approach has then been used for recovering feature-to-code mappings. This results in a novel approach exploiting the synergies between program analysis and diffing techniques to reveal feature-to-code mappings for configurable software systems.

The ICD approach specifically contributes to research goals RG2 and RG3 because it reduces the source code complexity of customized product variants during maintenance and supports the integration of customer-specific extensions into the SPL.

Contribution 3 – Configuration-Aware Change Impact Analysis

Variability-aware program analysis techniques have been proposed for analyzing the space of program variants. Such techniques are highly beneficial, e.g., to determine the potential impact of changes during maintenance. Chapter 5 presents the *configuration-aware change impact analysis (CA-CIA)* approach for determining possibly impacted products when changing source code of a product family. The approach supports engineers who are adapting specific product variants after an initial preconfiguration. The CA-CIA approach also uses the CSDG to perform program slicing based change impact analysis (CIA). However, in contrast to traditional CIA, CA-CIA leverages variability information to improve the precision and provides information about involved product variants.

The evaluation shows the benefits and the performance of the approach using the KePlast product line of the industry partner. It shows that the approach provides more precise results than existing CIA approaches and it can be implemented using standard control flow and data flow analysis.

The CA-CIA approach contributes to research goals RG1 and RG2 because it considers variability information when computing the possible change impact for an intended modification and it allows to compute a change impact for a specific configuration.

Contribution 4 – Compositional Change Impact Analysis for Configurable Software

Software systems are usually designed and implemented in a modular way to address challenges such as complexity, multi-language systems, distributed development, and continuous and long-term evolution. Analyzing large-scale software systems can lead to performance issues, resulting in huge dependence graphs and long analysis times. Therefore, the CA-CIA approach has been extended for supporting a compositional analysis where CSDG are built for single modules and then can be composed together in the same way as composing modules.

Thus, the *configuration-aware modular CIA (CAM-CIA)* approach exploits the modularity of large-scale systems to first perform program analysis for individual modules, and later compose the pre-computed analysis results. However, partitioning a CSDG is not straightforward as it carries presence conditions representing variability. The approach uses placeholders at module boundaries that are resolved when composing the pre-computed CSDG modules during configuration-aware program analysis. The approach is particularly useful in the context of product lines when product variants are derived by composing modules depending on specific customer requirements.

The evaluation investigates and shows the correctness of the CAM-CIA approach and its benefits compared to CA-CIA based KePlast product family.

The CA-CIA approach contributes to research goal RG4 since it enables to perform analysis on subsystems independently and then composing the results

Contribution 5 – Comparing Lifted and Delayed Variability-Aware Program Analysis

Two strategies have been proposed to make existing program analyses techniques variability-aware: (i) program analysis can be *lifted* by considering variability already in the parsing stage; or (ii) analysis can be *delayed* by considering and recovering variability only when needed. Both strategies have advantages and disadvantages, however, a systematic comparison is still missing.

Thus, an in-depth comparison has been conducted between the tools SPL$^{\text{LIFT}}$ and COACH!, which follow the lifted and the delayed strategy, respectively. It proved that the SPL$^{\text{LIFT}}$ is more accurate in most cases but COACH! is notably faster.

1.4 Publications

The following list summarizes the peer-refereed papers which contribute to this thesis.

ICSSP 2014 [Lettner14b] contains results of our in-depth interview case study. Our goal was to characterize the system of our industry partner and we therefore collected software ecosystem characteristics from literature and aligned these characteristics with findings from our case study.

SEAA 2014 [Lettner14a] presents further results of our interview case study. This paper investigates the evolution challenges in our industry partner's software system.

ASE 2014 [Angerer14a] is a doctoral symposium paper describing the goals and plan for this thesis.

SPLC 2014 [Angerer14b] presents the conditional system dependence graph and shows how it can be used to find inactive code on the basis of configuration.

ICSME 2014 [Linsbauer14] builds on [Angerer14b] and presents how to combine the ICD approach with clone detection for compile-time based product lines to be able to handle SPLs based on load-time configuration options.

ASE 2015 [Angerer15] presents our configuration-aware change impact analysis approach. It is based on program slicing techniques but uses the CSDG and propagates variability information.

SANER 2016 [Grimmer16] reports how we transform PLC programs to the object-oriented intermediate language Jimple while preserving analysis semantics.

INDIN 2016 [Prähofer16] discusses how to integrate the different methods developed in our research project to address the development and evolution challenges of the industry partner.

ICSME 2016 [Angerer16] describes how to modularize our configuration-aware change impact analysis to incremental and partial analysis, which is in particular helpful in multi-language systems.

SoSyM 2016 [Rabiser16b] presents the FORCE modeling approach for multi-purpose multi-level feature modeling and an Eclipse-based tool implementing this approach. The approach uses the proposed program analysis methods presented in this thesis.

Chapter 2

Background

This chapter contains the scientific background for this thesis and covers following areas: the area of variability including variable software, configurable software, software product lines, and clone-and-own approaches. The area of program analysis including static program analysis, program slicing, and change impact analysis. And the combination of these areas known as variability-aware program analysis.

2.1 Variability

Software variability, i.e., the property of a software system allowing its customization to different application scenarios, is regarded as a successful implementation strategy for increasing software reuse and for developing highly customized solutions [Svahnberg05]. It can be applied where requirements are similar and therefore developers have to implement similar solutions [Lettner14b]. However, as requirements are not really identical but vary, the implementations also have to vary.

A good practice to manage such differences in the implementations is to have a common base and just vary the implementation where it is necessary [Clements02]. For implementation of software variability, there are different implementation concepts. For example, developers may use preprocessor directives [Liebig10], custom-developed configurators [Lettner13], aspect-oriented programming [Kiczales97], delta-oriented programming [Schaefer10], feature-oriented programming [Apel09], or load-time configuration options [Lillack14] to name but a few.

For example, the C preprocessor (CPP) is a popular tool which adds an additional step to prepare the source code for the compiler [Kernighan88]. The CPP also provides a directive that allows to include or exclude source text depending on a given configuration. Therefore, it is a simple but powerful mechanism for implementing variability [Liebig10]. For example, Listing 2.1 is a small program using CPP directives to implement an optional feature A. By setting the directive, it is possible to include or exclude the dependent source code, thus realizing variability in software.

However, the use of preprocessors also has major drawbacks. For example, the

```c
void foo()
{
  int x = 0;
  x = x + 1;
#ifdef A
    x = x * 2;
#endif

#ifdef !A
    x = x / 2;
#endif
  printf("%d\n", x);
}
```

Listing 2.1: The source code of a SPL using preprocessor directives to implement variability.

CPP introduces directives that also appear in the source code, i.e., the CPP defines a language for metaprogramming [Liebig10]. Therefore, analysis tools for the C programming language can only work on processed source code or need to understand CPP directives [Kästner11]. Furthermore, while CPP is well integrated into the majority of the C compiler tool chains (e.g., the GNU Compiler Collection), other preprocessors are not related to a programming language at all and integration must be done manually [Kästner12].

Another widespread technique for implementing variability is to use configuration options that are loaded from a resource, e.g. a file, for controlling conditional execution [Lillack14]. This has the advantage that no additional techniques are required and variability can directly be implemented in the programming language. The value of a load-time configuration option is determined at the time of the program start, e.g., in an initializing phase, and then remains constant during program execution. This thesis therefore makes an important assumption and defines that configuration options' values must at least be bound at load time. However, the configuration option's value is of course stored in a program variable and may propagate in the program. As an example, consider the tiny configurable program in Listing 2.2. The program loads properties from a file named `conf.prop`, tests if option *logging* is enabled, and then stores the result in variable `loggingEnabled`. If option *logging* is enabled, a second option named *logToSock* is tested and thus the first configuration option already influences the control flow. Both values are then used to initialize the logging subsystem, however, the values are not independent.

Load-time configuration options can be seen as a way between compile-time and run-time configuration. Syntactically, they are indistinguishable from run-time configuration because a value is loaded from a source and the value is stored in program variables.

```
class Main {
  static Properties prop = Properties.load("conf.prop")

  public static void main(String[] args) {
    boolean log = "on".equals(prop.getProperty("logging"));
    boolean logToSock = false;
    if (log) {
      logToSock = "on".equals(prop.getProperty("logToSock"));
    }
    LogManager.initialize(log, logToSock);
  }
}
```

Listing 2.2: Influence of configuration options on program execution.

Semantically, load-time configuration option are close to compile-time constants because the value is loaded in the startup phase of the program and remains constant over program execution time. The major difference to compile-time configuration is that the configuration value can be used in the program as every other value. It can be used in expressions, passed as parameter, published as global variable and so on. Another issue of load-time configuration options also is that the shipped applications always contain the whole code with all the options and there is no separation between configuration code and application logic.

2.2 Software Product Lines

A software product line (SPL) is a systematic approach for managing the variability of software [Clements02] [Czarnecki00] [Pohl05]. It is defined as *a set of software-intensive systems sharing common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way* [Clements02]. SPLs identify and manage common functionality relevant for most customers and then support deriving customized solutions based on the customers' individual requirements. The SPL approach consists of two life cycle phases: domain engineering and application engineering [Pohl05]. In domain engineering, the commonality and variability of the SPL is defined and reusable artifacts are constructed. In application engineering, a concrete application satisfying the specific application requirements is realized by exploiting the commonality and variability of the SPL.

SPLs can be implemented in several ways, e.g., using compositional approaches [Kästner08] or using an ad-hoc approach like clone-and-own [Rubin13] [Linsbauer13]. Compositional approaches have been developed to provide an automated process for composing new product variants. In order to derive a product variant, one needs to select the software artifacts, e.g. features, aspects, or deltas, to be included. A composer then generates the variant by putting the selected artifacts together according to the specific composition rules.

Such compositional approaches assume that a system has been designed as SPL from its inception and that features have been decomposed to components already. However, legacy software systems commonly have not been created with an SPL perspective. Companies are thus only slowly migrating legacy software systems to SPLs as existing systems cannot be fully re-implemented using a compositional approach [Linsbauer13]. Furthermore, some programming languages used in industrial practice have only limited support for using compositional techniques. As a result, products are often developed using a clone-and-own reuse approach by adapting existing solutions [Rubin13] [Linsbauer13]. Clone-and-own is the process of using an existing product variant as a basis for a new solution and then adapting the clone such that it fits the new requirements [Dubinsky13]. Thereby, functionality is often added using load-time configuration options that enable this functionality. Cloning and adapting variants has the major advantage that this process is very intuitive, there are no initial costs, and existing variants are not affected by modifications. However, maintaining a clone-and-own product line quickly becomes very time consuming and error prone [Linsbauer16]. It is even difficult to determine the variants affected by a change.

2.3 Static Program Analysis

Static program analysis is a technique for acquiring information about the structure but also run-time behavior of a program without executing it [Nielson99]. Its main benefit lies in improving the quality of code in early development stages by providing means to reveal inadequate code constructs (”code smells”), violations of programming guidelines, and potential defects [Louridas06]. Numerous techniques and tools are available for general purpose programming languages like C/C++, Java, C# as well as for many others [Payet12]. Moreover, several empirical studies confirm its valuable contribution to software quality [Zheng06] [Ayewah08].

Prominent examples for concrete analyses are control-flow analysis, data-flow analysis [Nielson99], abstract interpretation [Cousot77], program slicing [Weiser81], and change impact analysis. Control-flow and data flow analyses are techniques originally developed in context of compiler technology [Muchnick97]. Control-flow analysis determines how the program can be executed, i.e., which program paths may occur. Data-flow analysis is about understanding which values can be created in a program and how the values can be manipulated and used [Allen01]. Typical examples of data flow analysis methods are reaching definitions, live variables and available expressions. Abstract interpretation extracts information about the behavior of a program by partially

```
 1  BEGIN
 2  READ(X, Y)
 3  TOTAL := 0.0
 4  SUM := 0.0
 5  IF X <= 1
 6  THEN SUM := Y
 7  ELSE BEGIN
 8  READ(Z)
 9  TOTAL := X*Y
10  END
11  WRITE(TOTAL, SUM)
12  END.
```

Listing 2.3: The original example as presented by Weiser [Weiser81].

executing the program using abstract operations on abstract values.

Program slicing is a technique for reducing a program to the subset of statements, i.e., the slice, which faithfully represents a specific program behavior [Weiser81]. The method is based on the observation that for producing a particular program behavior, often only a subset of a program is required. The goal usually is to reduce the effort required to understand and maintain the program by only having to consider a part of it. For example, consider the program in Listing 2.3, the original example from Weiser's paper [Weiser81]. Assume that we are interested in the outcome of value `TOTAL` in line 11. Computing a slice starting from this expression results in a smaller program denoted by the highlighted lines in the listing. Some of the statements have no influence on the variable's value and can therefore be ignored when investigating the problem. Therefore, program slicing is a program analysis technique that computes a set of program statements, i.e., the program slice, that may affect the values at some program point, i.e., the slicing criterion.

The original slicing algorithm proposed by Weiser [Weiser81] uses the control flow graph (CFG) of a program. Ottenstein et al. [Ottenstein84] then formulated slicing as a graph reachability problem on the procedure dependence graph (PDG) but this method was limited to intraprocedural slicing. Furthermore, Horwitz et al. [Horwitz90] introduced the SDG and a traversing algorithm for finding interprocedural slices. In general, using reachability algorithms on dependence graphs is the most popular way for computing slices [Xu05].

Change impact analysis (CIA) is the process of determining the potential effects of a proposed modification in the software [Bohner02]. In the context of this thesis, a modification of the software means a modification of the source code and the effect of a modification is a set of impacted statements. Since the exact impact of a change is again hard or impossible to compute, change impact analysis approaches just compute a possible impact. There are many CIA approaches and they can be classified into

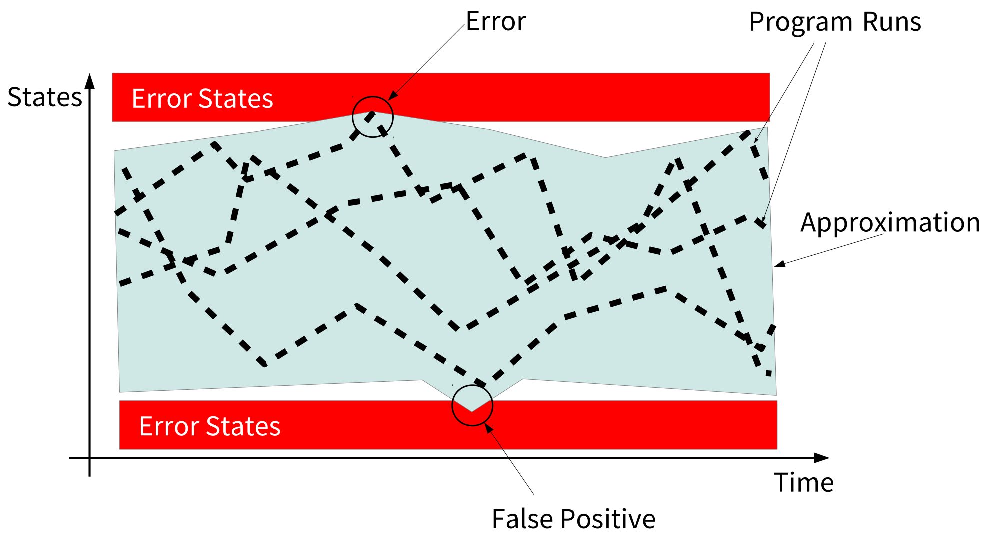


Figure 2.1: Illustration of state space overapproximation [Cousot05]. The dashed lines represent real program runs. The green area indicates legitimate program states and the red areas indicate error states.

guided and unguided techniques, techniques using heuristics, approaches using static or dynamic approaches and combinations of several approaches. One approach for change impact analysis is based on program slicing techniques and this thesis pursues slicing-based CIA techniques.

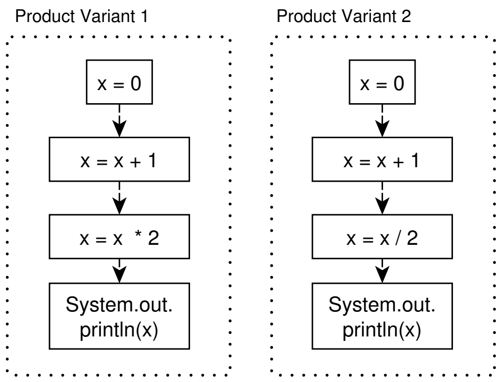
The core idea of static program analysis is to explore all possible states of the program based on its source code. However, according to *Rice's Theorem* [Rice53] all non-trivial questions about the behavior of a program are undecidable. Therefore, static code analysis has to work with approximations of the program [Nielson99]. The approximation of a program can be explained as follows: executing a concrete program produces a sequence of program states. The sum of all possible executions forms the set of reachable states. An approximation of the program is a simplification of the program's semantics and has therefore a different set of reachable states. The goal is to build an over-approximation which has at least the reachable states of the concrete program but may have additional ones. An analysis is *sound* if it works on an over-approximation of the program because then the possible states, i.e., the possible behavior, of the program is covered. The *precision* of the approximation is determined by these additional states, i.e., fewer additional states increase precision.

Figure 2.1 illustrates the relationships between the reachable states, the approximation and the error states. The red areas at the top and at the bottom of the diagram depict the error states. The dashed lines indicate the program runs which are actually feasible in the real program. If a program run reaches a state within a red area, the program

```
public void foo() {
  int x = 0;
  x = x + 1;
//#ifdef A
    x = x * 2;
//#endif

//#ifdef !A
    x = x / 2;
//#endif
  System.out.println(x);
}
```

(a)



(b)

Figure 2.2: A very small SPL with two possible product variants.
(a) A preprocessor-based SPL.
(b) Data dependence graphs for the two possible product variants.

is erroneous. The surrounding blue area indicates an approximation of the program's behavior. Obviously, the approximation is an over-approximation because it covers all reachable states of the real program but also contains states that are actually not reachable by the real program. If the approximation contains states of a red area (error states) but the real program does not, the analysis will signal an error in the program which effectively does not occur. This is called a *false positive*.

2.4 Variability-Aware Program Analysis

This section describes the fundamental concept of variability-aware program analysis which is the basis for the CA-CIA approach in Chapter 5.

Using program analysis techniques during development and maintenance of SPLs requires to consider all possible product variants [Brabrand12]. This is necessary because every single possible product variant must be considered to ensure that the whole SPL is correct. Unfortunately, the number of possible product variants grows exponentially with the number of available options [Thüm14]. For example, the small product line in Listing 2.2 allows to generate two product variants, one with feature A and one without it. A developer performing a reaching definitions analysis to ensure that the data flow is correct for all possible product variants needs to generate every possible product variant and perform the analysis for each variant individually. Figure 2.2b shows the two possible data dependence graphs (DDGs) resulting from a reaching definitions analysis. Obviously, the DDGs are very similar since they differ only in one node. Looking at this example suggests that it is straightforward to do the analysis only once for all parts that are in common. Therefore, researcher developed variability-aware program analysis

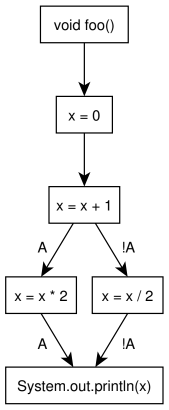


Figure 2.3: The control flow graph for the product family shown in Figure 2.2

that use the same principle as SPLs for analyzing SPLs [Brabrand12] [Liebig13]. The idea is to do the analysis for all common parts just once and only diverging the analysis if the program diverges.

This is achieved by annotating intermediate analysis data structures and results with variability information [Walkingshaw14]. In the case of a reaching definitions analysis, the intermediate results are the definition sets. For example, consider again the program in Listing 2.2a. The definition sets `{(x, 0)}` and `{(x, x + 1)}` flow out of statements in line 2 and line 3, respectively, and are independent of features. However, the definition set flowing out of statement in line 5 depends on feature A being enabled. Therefore, the definition set `{(x, x * 2)}` has to be annotated with feature A, which is written as `{((x, x * 2), A)}`. In contrast, the definition set flowing out of statement in line 9 is only valid if the feature A is disabled, i.e., it is `{((x, x / 2), !A)}`. Thus, by adding conditions to definition sets representing the product variants, the intermediate results are made variability-aware.

The core concepts of variability-aware program analysis are late splitting and early joining [Liebig13]. Late splitting means that the analysis is performed without variability until variability is encountered. Early joining is the concept of collapsing intermediate analysis results as soon as possible, i.e., if data flow from different product variants reaches a destination that is common for several product variants again. Figure 2.3 illustrates the CFG of the program in Listing 2.2a. The CFG is variability-aware because it contains the information about which edge is valid in which product by using Boolean presence conditions. When computing the reaching definitions on the CFG in a variability-aware manner, the analysis will not split until variability is encountered, i.e., if edge from node `x = x + 1` to `x = x * 2` or edge from node `x = x + 1` to `x = x / 2` are visited. This is the principle of late splitting. Furthermore, if node

`System.out.println(x)` is visited, the incoming information is merged by combining the presence conditions. In this way, the number of entries in the reaching definition sets is kept low and the complexity of the variability is moved to the presence conditions. Of course, this looks like complexity is just shifted to another place but in practice, handling Boolean formulas is way more efficient than having explicit entries in the reaching definitions sets.

Variational data structures efficiently represent variability in data and thus enable variability-aware computations [Walkingshaw14]. In the context of this thesis, variational data structures are required to represent the variability of a software system. For example, to store the artifacts of a SPL it is necessary to store the information about when to include an artifact in a product. Furthermore, variational data structures are often used for variability-aware program analysis. A well known example for such data structure is the variational AST which is an important data structure for program analysis. The tool TypeChef [Kästner11], for instance, parses preprocessor-annotated source code and represents the variability of the source using a variational AST.

This thesis also describes a variational data structure, the CSDG. Since the CSDG has been developed and published before the general concept of variational data structures has been published by Walkingshaw et al. [Walkingshaw14], it is named a bit different.

In this thesis, the term *conditional* is used in data structures to denote that the data structure might look different depending on external conditions.

Current variability-aware program analysis techniques incorporate variability information from the very beginning. This kind of analysis is usually referred as *lifted analysis* [Brabrand12]. The concept of analyzing the whole space of possible program variants at once has the advantage that it is as precise as analyzing all program variants one after each other. However, it also has drawbacks as discussed previously, i.e., a lifted analysis cannot fully handle load-time configuration options. We therefore proposed the analysis concept called *delayed variability* analysis (cf. Section 1.1).

Figure 2.4 opposes the two analysis strategies. The delayed variability analysis works by performing an analysis completely variability-oblivious, i.e., ignoring any variability in the first place, and then recovering the variability and augmenting the results. This has the advantage that just one analysis is performed. In contrast to this, a lifted analysis conceptually performs as many analyses as program variants are possible. However, it is possible to optimize lifted analysis by sharing analysis results. The optimization exploits commonalities and just forks the analysis where it is required due to different variants (cf. early joining and late splitting). Nevertheless, it may be the case that the lifted analysis needs to split early and cannot join early which leads to many explicit intermediate results. This situation is avoided in a delayed variability analysis because the analysis is done just once regardless of variability and can lead to

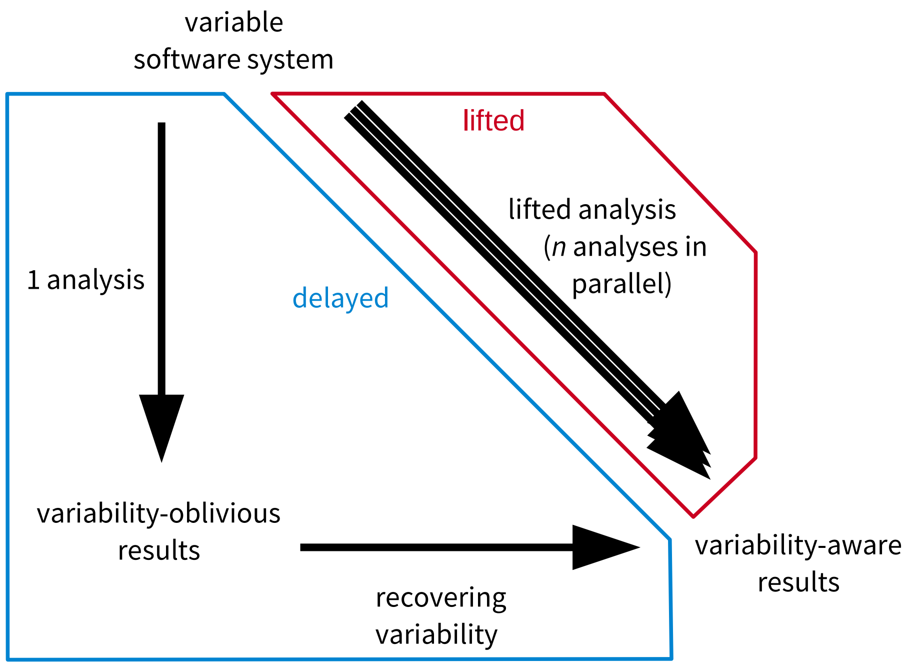


Figure 2.4: The delayed variability principle.

better run-time performance. However, static program analysis always has to make a tradeoff between run-time performance and precision. The drawback of this strategy is the loss in precision because the delayed variability analysis needs to overapproximate situations to be sound. Chapter 7 provides a detailed comparison and discussion about the advantages and drawbacks of the two analysis strategies.

Furthermore, the delayed variability concept assumes a graph structure that represents the execution semantics appropriately. The CSDG, for example, is such a graph structure satisfying this requirement. The delayed variability concept also allows to incorporate the specific semantics of a dependence edge in the graph.

Chapter 3

Conditional System Dependence Graph

This chapter introduces the *conditional system dependence graph (CSDG)*, the basic data structure for the configuration-aware program analysis methods presented in this thesis. The CSDG is based on the system dependence graph (SDG) that has been introduced by Horwitz et al. [Horwitz90] for globally representing control and data dependencies in a program. The CSDG extends the SDG by introducing presence conditions for representing the variability of a program and is therefore a variational data structure (cf. Section 2.4).

This chapter is based on [Angerer14b] and [Grimmer16] and is organized as follows: Section 3.1 introduces the SDG. Then, Section 3.2 explains how the SDG is extended to represent variability resulting in the CSDG. Further, Section 3.3 describes how to detect variability in a program and how to insert variability information into the CSDG. Section 3.4 describes how to deal with expression that mix load-time values and run-time values. Section 3.5 defines the semantics of the CSDG, in particular, the semantics of the variability information. Section 3.6 gives an overview of the tool chain for building the CSDG. Finally, Section 3.7 reports numbers on a performance evaluation for building the CSDG.

3.1 System Dependence Graph

A SDG is a directed graph representing different kinds of dependencies between program elements. It usually represents control-flow and data-flow dependencies, but other types such as definition-use dependencies are also possible [Horwitz90]. In this thesis the following node types are used for representing concrete or abstract program elements in the SDG (cf. Figure 3.1):

Method nodes represent methods, procedures, functions or any other type of callable program unit. In the following no distinction is made between those types of program elements but the term *method* is used as a synonym for all types of callable program units. Thus, method nodes represent the entries of callable program elements. Further, they allow grouping parameter nodes and statement nodes. Figure 3.1 shows the SDG

```
EXTERNAL BOOL config_double;
INT global := 0;

PROCEDURE main
  INT a := foo(bar());
  PRINT("result = %d", a);
END_PROCEDURE

PROCEDURE foo(INT p0) : INT
  IF config_double THEN
    RETURN p0 * 2;
  ELSE
    RETURN p0;
  END_IF
END_PROCEDURE

MODULE M1
  PROCEDURE bar : INT;
    global--;
    IF global < 0 THEN
      global := 0;
    END_IF
    RETURN global;
  END_PROCEDURE
END_MODULE

MODULE M2
  PROCEDURE bar : INT
    global--;
    RETURN global;
  END_PROCEDURE
END_MODULE
```

Listing 3.1: Sample program for illustrating the CSDG.

for the sample program in Listing 3.1. There are method nodes for the procedures `main`, `foo` and the two variants of `bar` in Module `M1` and `M2`, respectively. Additionally, the artificial method node `root` is used to represent the entry point of an application and acts as a container for global variables.

Statement nodes represent the statements in methods and always belong to one method node.

Call nodes are used to represent method calls, i.e., call nodes are special types of statement nodes. They are particularly important in the SDG as they link statement nodes to method nodes. Figure 3.1 shows call nodes for the calls to methods `foo` and for both versions of `bar`.

Formal parameter nodes are used for representing any data dependencies between a methods and its environment. The return values of methods and access to global variables are also represented as formal parameter nodes. Figure 3.1 contains formal parameter nodes (with black background) for the method parameters, return values,

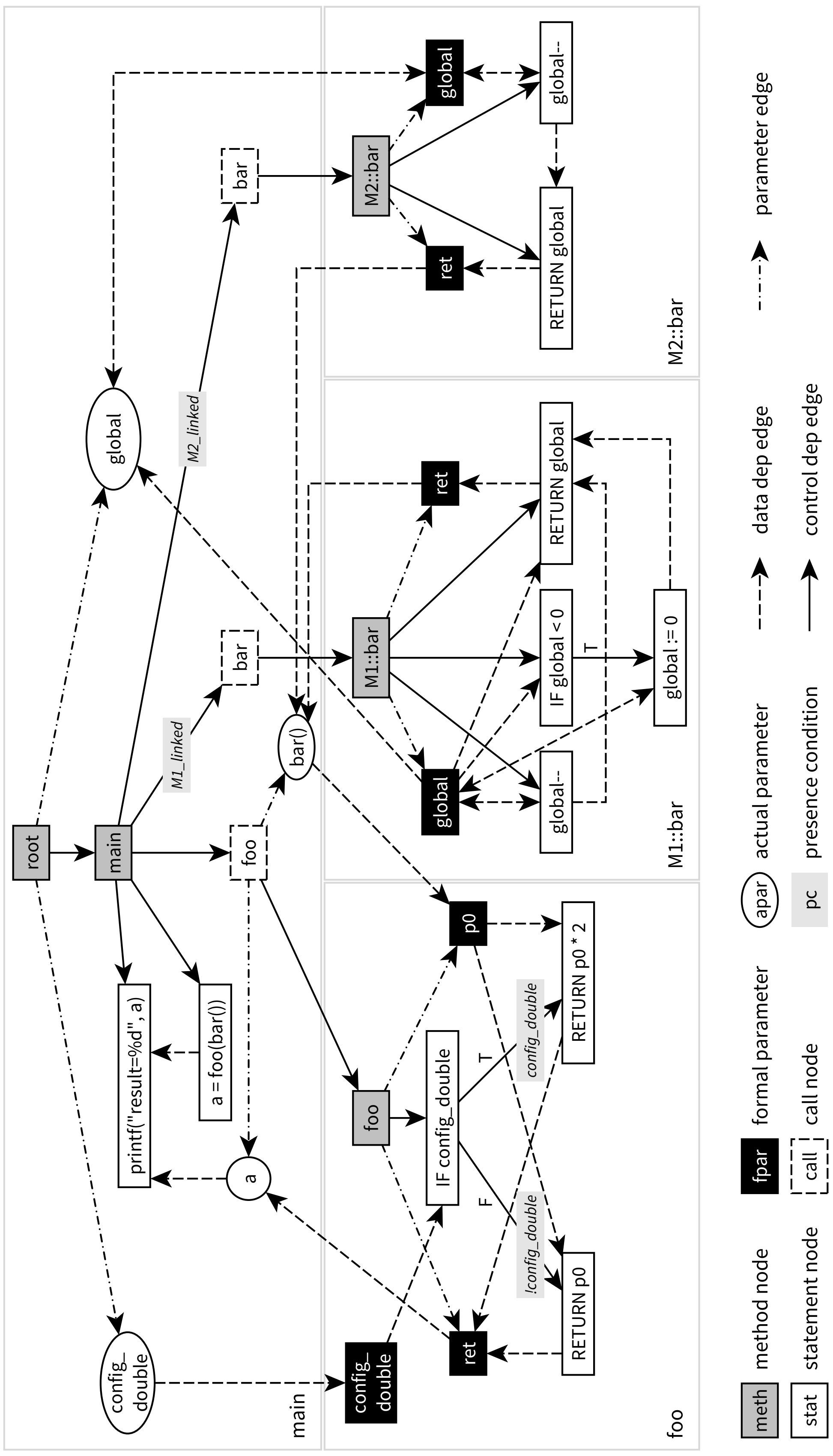


Figure 3.1: The CSDG for the sample program in Listing 3.1.

and the global variable `global` accessed in the two versions of procedure `bar`.

Actual parameter nodes represent expressions for actual parameters in method calls.

The SDG contains the following types of edges for representing the various types of dependencies between program elements:

Control edges link method nodes with statement and call nodes directly contained in the method. Control edges represent that a statement or call will be executed if the method is called. Furthermore, conditional statements are represented as control edges labeled with true (T), to connect the condition with all the nodes of the then branch, and edges labeled with false (F), to connect the condition with the nodes of the else branch. For example, the statement `a := foo(bar())` in Listing 3.1 is executed unconditionally as soon as the procedure `main` is executed. Therefore, the corresponding statement node in the SDG (cf. Figure 3.1) is directly connected to the corresponding method node with a control edge. The statement `RETURN p0 * 2` is only executed if the condition `config_double` evaluates to true. This is represented by the control edge labeled with T between the corresponding statement nodes in the SDG. On the other hand, the statement `RETURN p0` is only executed if the same condition evaluates to false. This is represented by the control edge labeled with F.

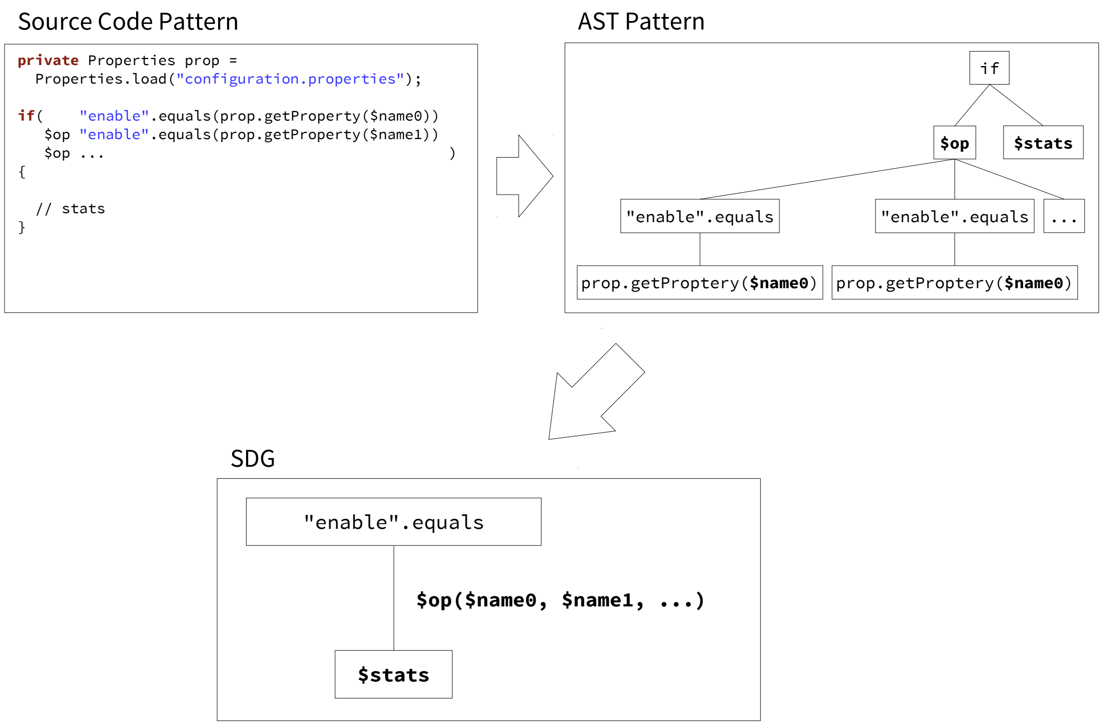
Parameter edges connect method nodes with all formal parameter nodes used by the method. Analogously, there are parameter edges from a call node to the actual parameter nodes. Parameter edges are represented with dashed-dotted lines in Figure 3.1.

Data dependence edges show data dependencies between parameter or variable nodes and statement nodes in the SDG. Data dependence edges between actual and formal parameter nodes show parameters passing during method calls. Data dependence edges between statement nodes represent data-flow dependencies. In Figure 3.1 data dependencies are shown with dashed lines.

3.2 Presence Conditions

A CSDG is built by annotating edges in the SDG with presence conditions. Presence conditions are Boolean formulas representing the variability of system, i.e., the Boolean formulas describe a set of valid product variants [Kästner11]. An edge is valid for a specific configuration if the condition is satisfied with respect to that concrete product configuration. Hence, a condition defines the *presence* of a certain edge for specific product configurations.

Dependent on the type of edge, the presence conditions can be interpreted as follows: if a control dependence edge is annotated with a presence condition, the execution of the program element will only occur in a configuration satisfying the presence condition. Analogously, a presence condition attached to a data dependence edge means that the



```
private Properties prop =
  Properties.load("configuration.properties");

if(    "enable".equals(prop.getProperty($name0))
   $op "enable".equals(prop.getProperty($name1))
   $op ...                                    )
{

  // stats
}
```

Figure 3.2: CSDG condition extraction.

data flow only exists in the product variants satisfying the presence condition.

Consider the configuration variable `config_double` in Listing 3.1: There will be presence conditions on the edges connecting the statement node `config_double` in procedure `foo` with nodes `RETURN p0*2` and `RETURN p0` as shown in the CSDG in Figure 3.1. Further, the modules `M1` and `M2` contain two alternative implementations of procedure `bar`. Depending on which module is actually linked, one of the variants will be called. Therefore, there are presence conditions `M1_linked` and `M2_linked` on the two respective edges.

3.3 Representing and Extracting Variability

There are different ways how variability can be implemented in a program and the presence conditions are used for expressing and abstracting from different variability implementation concepts. For example, variability can be implemented using conditional statements testing configuration settings (load-time configuration options) [Lillack14]; aspect-oriented programming [Kiczales97] can be used to inject additional behavior with cross-cutting implementations; or module link configurations determining the modules to be used in case alternative modules are available. Section 4.1.3 shows

```
switch(movementAxes)
{
case 1: /* code for 1-dimensional robot */
        break;
case 2: /* code for 2-dimensional robot */
        break;
case 3: /* code for 3-dimensional robot */
        break;
default: /* error */
}
```

Listing 3.2: Source code sample of the industry partner handling a non-Boolean configuration option.

examples for variability mechanisms used by the industry partner.

Furthermore, compile-time variability [Kästner11] allows implementing variability by selecting and composing source code snippets, e.g., in C/C++ using preprocessor directives. Such compile-time variability mechanisms, however, may modify the source code in a non-structured way, i.e., the composed code may violate the language syntax. An approach to cope with preprocessor approaches is by transforming compile-time to load-time variability. However, this issue is not pursued further in this thesis and the reader is referred to von Rhein et al. [vonRhein16], which describe a comprehensive approach for handling compile-time variability.

In the approach pursued in this thesis, Boolean formulas are used for defining presence conditions. Further, for representation and handling of presence conditions, binary decision diagrams (BDDs) are employed which allow a concise encoding and provide highly efficient operations for manipulating Boolean formulas. However, this comes with the major restriction that the abstracted value of a configuration option can only be true or false. Therefore, an approach for encoding arbitrary configuration conditions is needed.

In real-world programs, configuration is often not only of strict Boolean nature. For example, the industry partner also uses numeric values for configuration options. Their robot control software is able to handle robots with one, two, or three movement axes and this is specified using the value domain $\{1,2,3\}$. The source code testing the configuration option `movementAxes` is shown in Listing 3.2. To cope with this situation, predicates for testing for specific configuration values are introduced, e.g., `IS_THREE_-AXIS_ROBOT(movementAxes)` for testing for a 3-dimensional robot. This, however, gets infeasible when value domain are huge, e.g., the Integer value domain, because this would lead to 2^{32} possible predicates. However, the industry partner does not use huge value domains and often just enumerations with a few values are used for configuration, which can easily be represented by Boolean predicates.

Furthermore, relations between predicates are lost in abstraction. For example, code testing `if(movementAxes < 0 && movementAxes > 3)` is unsatisfiable. By abstracting the conditions to predicates `LESS_THAN_ZERO(movementAxes)` and `GREATER_THAN_THREE(movementAxes)` one looses the information that both cannot be true at the same time and the conjunction will always be false.

Variability information is extracted from the source code and thus the variability mechanism used in a software system must be known. Recall that variability may be implemented in different ways, e.g., using preprocessor directives or configuration options. The approach presented in this thesis specifically targets to support *load-time configuration options*. Thus, it is assumed that programs load configuration options in the start-up phase and then store them in program variables, which stay constant during execution.

For finding configuration options, the source code of a system is analyzed to find statements loading (and possibly combining) configuration options and store them in program variables, i.e., statements that assign *configuration conditions* to *configuration variables*. Each configuration condition then becomes a *seed condition* attached to a node in the SDG and the CSDG represents the configuration options in the form of Boolean *configuration conditions* indicating which options are enabled.

Finding seed conditions depends on the way how variability is implemented. For instance, the industry partner's developers use a dedicated interface to test if a configuration option is enabled. Therefore, in our case study mining seed conditions is done by a structural analysis of the AST of the program. Figure 3.2 illustrates the process: first, a source code pattern for matching a specific variability implementation is defined. In the example, the pattern defines the use of method *java.util.Properties.getProperty(String)*. If the method call occurs in a conditional test of a branch statement, the name of the configuration option is extracted and used as seed condition. The pattern also allows logical combinations of several configuration options and uses pattern variables `$op` for logical operators. Second, the source code pattern is transformed into an AST pattern and the actual matching is done on the AST. The seed condition is then built from the matching logical operator `$op` and the matching configuration option names `$name0`, `$name1, ...`. This seed condition is finally attached to the outgoing edge of the SDG node representing the branching statement. The negation is attached to the edge of the else branch, if it exists.

Abstracting and extracting variability information may of course lead to information loss. As it is the case in static program analysis, however, this just leads to less precision and the analysis still remains sound.

In the case of abstracting non-Boolean value domains, the abstraction is sound and the precision decreases because more product variants are allowed. Recall the above example introducing the two predicates `LESS_THAN_ZERO(movementAxes)` and

```
if(movementAxes == 1) {
  // ...
}
if(movementAxes == 2) {
  // ...
}
```

Listing 3.3: A small code snippet demonstrating that predicate abstraction leads to overapproximation.

```
int value = 12345;
if(SomeCustomConfInterface.isEnabled("log")) {
  Logger.info("Value is: " + value);
}
```

Listing 3.4: A small code snippet demonstrating that variability extraction may lead to overapproximation.

`GREATER_THAN_THREE(movementAxes)`. As mentioned previously, the conjunction of these two predicates is false, however, the abstraction looses the dependency between the predicates. Listing 3.3 illustrates this situation. If the first branch statement is entered, the second branch statement will not be entered. However, the abstraction using predicates does not know about this and for the analysis, both branches may be entered. If an analysis assumes that both branches can be entered, it also assumes that more program states are possible as can be reached during execution of the program. This is exactly the definition of overapproximation.

In the case of variability extraction, information may be lost if variability implementation patterns are not detected. Again, assume there is a branch statement testing a configuration option which is not detected. The analysis still remains sound if this information is missed because it is then assumed that the branch statement and the consequent statements are valid for all product variants using Boolean formula *true*. Listing 3.4 provides an example for this situation. Assume the call `SomeCustomConfInterface.isEnabled(...)` tests a load-time configuration option not recognized by the analysis tool. Then the presence condition will default to *true* at the corresponding control dependency indicating that the log statement will be executed in every configuration. Therefore, every program variant again has more program states which is again the definition for overapproximation (cf. Section 2.3).

3.4 Mixing Load-time and Run-Time Values

The value of load-time configuration options is stored in common program variables which is then called a configuration variable. However, this means that the configuration

```
1  static Properties prop = ...;
2  static boolean c0 = "enable".equals(prop.get("option0"));
3
4  public static void main(String[] args) {
5    boolean x = ...;
6    if(c0 && x) {
7      foo();
8    }
9    if(c0 && !x) {
10     foo();
11   }
12 }
```

Listing 3.5: A program mixing load-time and run-time values in branch expressions.

variable can be used in branch expression together with every other program variable. In Listing 3.5 variable `c0` is a configuration variable and `x` is a common program variable. The two different kinds of variables are used in the expression of the branch statements.

In order to keep the approach sound, it is necessary to ignore such expressions or in other words, to abstract to presence condition *true*. It can, however, make sense to defer this conservative abstraction for two reasons:

(i) The unknown value of the run-time variable could be eliminated by a negation.

(ii) The information that code partially depends on configuration can be useful.

To overcome the problem of unsoundness, a unique placeholder variable represents the run-time value. For example, instead of extracting the presence condition *true* from the branch statement in line 6, it is possible to use formula $c0 \wedge u_x$ whereas u_x represents the unknown run-time value of variable `x` at the particular program point.

To further illustrate the possible benefit of deferring the abstraction, Listing 3.5 has a second branch statement in line 9. Now, assume that method `foo` is just called from these two call sites. Then the method only depends on configuration option $c0$ because $c0 \wedge u_x \vee c0 \wedge \neg u_x$ is equivalent to just $c0$. Deferring the abstraction hence allows to eliminate the dependency on the run-time value and therefore preserves the variability information, which would otherwise be lost.

3.5 Semantics of the CSDG

In this section the semantics of the CSDG is formally defined. The definitions build on the semantics of the SDG, which are well-defined in literature [Ferrante87].

Definition 1. System Dependence Graph A SDG is a tuple

$$SDG = (V, E, T)$$

where V is the set of nodes of the SDG and $E \subseteq V \times V \times T$ are the edges and $T : E \rightarrow \{control, data\}$ is set of edge types.

The control-flow edges $e = (a, b, t) \in E \land T(e) = control$ represent the execution semantics. Intuitively, a node b is control dependent on a node a if the node *a decides* if b will be executed. The common example for this is an `IF` statement which decides if the statements in the body are executed.

The definition of control-flow dependency is based on by Ferrante et al. [Ferrante87] and is as follows:

Definition 2. Control-Flow Dependency There is a control-flow dependency $(a, b, t) \in E, t = control$ between nodes $a, b \in V$ if node a decides if node b is executed. We write $a \xrightarrow{ctrl} b$. More formally: $a \xrightarrow{ctrl} b$ if

1. There is a path p from a to b in the control-flow graph such that b post-dominates every vertex $v \in p, v \neq a$, and

2. b is not the immediate post-dominator of a

In other words, if a node has children in the control flow graph (CFG), then the node is a decision point. If a node b post-dominates a child of node a, then node b is control dependent on node a. In this case, node a then was the nearest decision point for the execution of b.

This definition uses the post-dominator relationship on the CFG which is defined as follows.

Definition 3. Post Dominance Relationship Node b *post-dominates* node a in the CFG if every path from a to $EXIT$ (excluding a) contains b [Ferrante87].

Based on the definition of the SDG, a formal definition of the CSDG is as follows.

Definition 4. CSDG A CSDG is a tuple

$$CSDG = (V, E, T, C, PC)$$

where V, E and T are the same as in the SDG, C is a set of Boolean configuration variables and PC is the presence condition function

$$PC : E \times \{True, False\}^C \rightarrow \{True, False\}$$

which for each edge determines a Boolean presence value which is dependent on the value assignments of Boolean configuration variables.

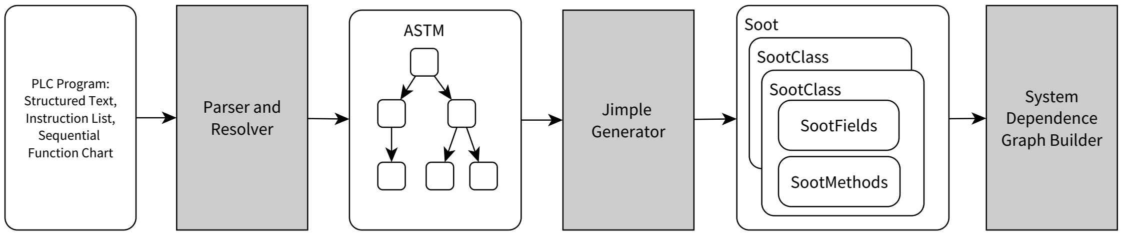

PLC Program: Structured Text, Instruction List, Sequential Function Chart
Parser and Resolver
ASTM
Jimple Generator
Soot
SootClass
SootClass
SootFields
SootMethods
System Dependence Graph Builder

Figure 3.3: Components and intermediate data structures of the analysis tool chain. First, a parser generating an AST representing the program was created. Second, a code generator that outputs Jimple code based on the AST for the Soot analysis framework was implemented. Third, Soot is then used to perform the necessary analyses to build the CSDG.

3.6 Implementation

We implemented a tool chain building the CSDG. Figure 3.3 provides an overview of the tool chain [Grimmer16]. However, the industry partner's software system KePlast is implemented in a programming language conforming to the IEC 61131-3 industry standard, a standard for programmable logic controller (PLC) programming. Unfortunately, the standard does not define a full programming language and therefore many manufacturer-specific dialects exist. Due to the diversity of the dialects, program analysis tools are hardly available for IEC 61131-3 based languages. KEBA also has its own language dialect and an analysis tool is also not available. It was therefore necessary to implement a full language front end for the analysis framework whereas the goal was to reuse existing technology to reduce the effort.

The implemented parser for the IEC 61131-3 language and used the CoCo/R parser generator [Wöß03] which generates an AST for an entire system. It uses OMG's ASTM [OMG07] as this reduces the effort for implementing an AST model for a programming language. The ASTM specification defines a generic part (generic ASTM) and a language-specific part (specialized ASTM). The generic ASTM (GASTM) provides AST node types commonly required in most imperative programming languages. This includes statement nodes for branches, loops, assignments, and so on. The specialized ASTM (SASTM) is supposed to be extended by the language implementer to support language-specific structures. In this case, the extension consisted mainly of node types representing definition types, i.e., variable definitions with special semantics, which are unique to KEBA's language dialect. Next, a code generator performs an analysis preserving transformation [Zhang06] that represents the control-flow and data-flow semantics of the program. The generator (cf. block *Jimple Generator* in Figure 3.3) therefore creates Jimple code based on the AST and also links the generated code to the AST nodes. Jimple is the intermediate language of Soot used to perform program analysis and optimization. Note, that the Jimple code does not represent the exact execution semantics of the original PLC program since this would require to fully implement KEBA's runtime environment. As soon as the Jimple code is available, the analysis capabilities of Soot are used for building the CSDG. This primarily involves a control-flow, reaching definitions, and pointer analysis.

As soon as the program to analyze is available as Jimple code, the analysis framework computes the intraprocedural PDGs based on intraprocedural control and data-flow information. This is the first step towards the creation of the SDG and PDGs represent the same information as the SDG but are limited to individual procedures. As an optimization, the analysis tool implements the algorithm by Harrold et al. [Harrold93] which computes PDGs directly based on the AST. However, this algorithm is limited to structured programs and therefore the generic but slower post-dominator approach as described by Ferrante et al. [Ferrante87] is available as backup procedure. The backup is necessary because unstructured transfer of control may happen, for example, when using the assembly-like *Instruction List* language. The SDG is then created by linking individual PDGs. Call nodes in the PDGs are then linked with the corresponding method nodes. The analysis framework also resolves data dependencies induced by indirect access via reference and pointer variables considering points-to sets. At this point, it is worth to mention that KEBA's IEC 61131-3 dialect does not allow any pointer arithmetic which is beneficial for the run-time performance of the analysis. Having all this information, the SDG can be completed.

3.7 Performance Evaluation

The performance evaluation had the goal to find out whether building the CSDG with the analysis framework is fast enough for using the tool chain during maintenance tasks. Therefore, the run-time performance evaluation used 34 different product variants from the KePlast platform. The set of products thereby covered the full bandwidth in terms of size in lines of code. We also included real customer variants and complemented these with freshly generated product variants. A dedicated goal was to cover every available feature of the product line at least once in a product variant. The conducted performance measurements determine the time required to build the AST, to generate Jimple code, and to generate PDGs and the SDG. Furthermore, the evaluation also measured the peak memory consumption (Java heap space). All performance runs were executed 10 times and the result is the median of the single values.

The tool chain is written in Java and we conducted the performance evaluation runs using a Java 8 HotSpot 64-Bit Server VM on Windows 7 running on a PC with an Intel Core i7 with 2.93 GHz and 16 GB DDR3-SDRAM. Table 3.1 shows the results of 8 representative products (PVs) of different size (the others show similar results). The

PV	Size [kLOC]	PM [GB]	AST [sec]	JCG [sec]	PDG [sec]	SDG [sec]	Total [sec]	SDG Size [k#Node]
1	53	1.5	2.8	2.5	11.2	14.2	30.7	106
2	60	2.1	3.9	3.2	20.1	22.3	49.5	164
3	81	1.8	3.9	4.0	22.4	19.5	49.8	187
4	98	3.3	4.9	5.4	20.9	42.8	74.0	188
5	205	7.3	10.2	16.5	107.6	36.8	171.1	416
6	253	8.0	12.2	23.5	130.5	49.9	216.1	604
7	265	9.4	12.7	24.4	135.4	53.8	226.3	626
8	302	9.5	14.9	29.8	163.9	62.9	271.5	708

Table 3.1: Results from the run-time performance evaluation. The size of the product variants (*PV*) is specified in lines of code (*Size*). *PM* indicates the peak memory consumption. *AST* represents the time needed for parsing and building the AST. *JCG* is the time for the Jimple code generation. *PDG* specifies the time required to build the PDGs including their instantiation, *SDG* is the time required for building the SDG. *Total* is the overall time needed. *SDG Size* shows the size of the resulting SDGs in number of nodes.

selected program sizes range from 53 kLOC to 302 kLOC. The total time for the whole analysis process is between 30 and 271 seconds, which is reasonable given the scope of the analyses. The resulting SDGs contain between 106k and 708k nodes. It can be observed that the time required to build the AST (column *AST*) and to generate the Jimple code (column *JCG*) is only about 1/6 of the overall time. Most time is spent on building the PDGs and the SDG, which uses the analysis results delivered by Soot.

Figure 3.4 relates the size and the total analysis time for 34 product variants (the selected product variants contained in Table 3.1 have circles as markers). We see a strong linear correlation (Pearson's correlation coefficient 0.982; p-value $< 2.2 \times 10^{-16}$). We found a similar strong correlation for the peak memory consumption (Pearson's correlation coefficient 0.983; p-value $< 2.2 \times 10^{-16}$). From these results it can be deduced that our implementation scales well to large-scale industrial applications.

3.8 Summary

This chapter introduced the conditional system dependence graph (CSDG) which is the basic data structure for the configuration-aware program analysis methods presented in this thesis. The CSDG is based on system dependence graph (SDG) and is built from the SDG by annotating edges with presence conditions. Presence conditions are Boolean formulas representing the variability of the system. An edge in the CSDG is valid if the associated presence condition is satisfied with respect to a concrete product configuration. Further, the semantics of CSDG were formally defined by relying on

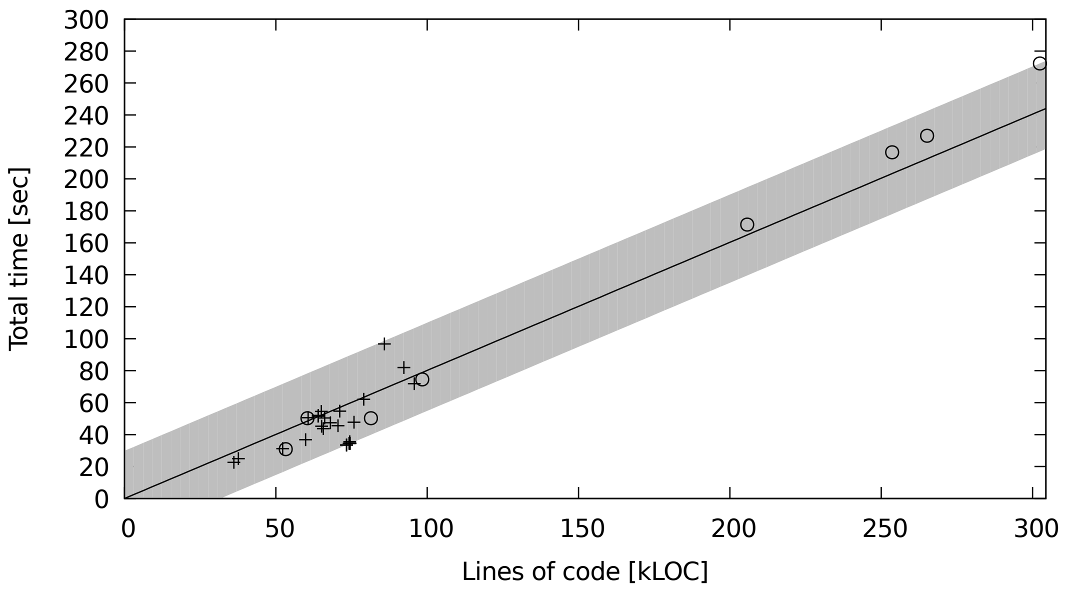


Figure 3.4: Timings tested on 34 product variants. The data points represented as circles are also shown in Table 3.1.

definitions of the SDG, control-flow dependency, and post dominance relationships.

This chapter also showed how variability mechanisms, i.e., the implementation of variability in the source code, are abstracted for extracting presence conditions. Abstracting and extracting variability information may lead to information loss, however, abstracting results in overapproximations and analysis results are still sound. Further, it was shown how to deal with non-Boolean configuration options and how to deal with situations where configuration variables are set together with normal program variables.

Finally, the implementation of the CSDG and the tool chain to build it were discussed. The run-time performance evaluation showed a linear correlation between the lines of code and the time as well as the memory consumption. Soot's analysis implementations obviously benefit from the restrictive nature of IEC 61131-3 languages, e.g., no dynamic memory allocations and no dynamic binding, because the algorithms converge quite fast. However, this is not automatically the case and implementing some optimizations like using a cache for the results of reaching definitions analysis is necessary. In summary, the performance results allow to conclude that the tool chain scales for industrial applications.

Chapter 4

Identifying Inactive Code and Recovering Feature-to-Code Mappings

One of the findings of the interview case study presented in Section 1.2 is that the industry partner first uses a custom-developed configuration tool for creating an initial product based on selecting the desired product features. This process is called *staged derivation* and is widely used in practice [Czarnecki05b]. Engineers then refine and adapt the source code to develop a solution for a particular customer. This means that developers customize the selected features by configuring their properties, similar to the idea of staged configuration [Lettner14b, Lettner13, Czarnecki04].

It is common in product lines that configuration parameters and configuration variables are evaluated by conditions or annotations in the code. However, a drawback of this approach is that the product variants created during application engineering contain a lot of code that is actually inactive, if considering the detailed configuration settings. Developing and maintaining product variants in such a way is cumbersome for application engineers as they have to read and understand a lot of source code that could be discarded for the specific product variant.

Feature-to-code mappings [Kästner08], feature location approaches [Eisenbarth03], or traceability techniques [Egyed10] help developers identifying source code related to particular features. However, the in-depth analysis of the industry partners systems showed that such mappings are frequently incomplete and do not consider the often complex source code dependencies of feature implementations. For instance, the code activated by a configuration parameter can have complex dependencies that need to be considered when identifying the inactive code for a certain configuration. The above mentioned approaches do not support a configuration-aware analysis of code dependencies.

This chapter presents a configuration-aware program analysis method based on the conditional SDG introduced in Chapter 3, which allows to identify code that is inactive in particular configurations. It is based on our publications [Angerer14b] and [Linsbauer14] and split into two parts: the first part (Section 4.1) presents our inactive code detection (ICD) approach for the automatic identification of inactive code for a concrete product variant based on configuration settings. The approach marks all code parts as inactive that cannot be executed in the current product configuration. Specifically, the first part provides three contributions: (i) An approach for identifying inactive source code based on configuration settings. (ii) A demonstration of the flexibility of the approach by tailoring and implementing it for an industrial product line. We show specific extensions for system-specific variability mechanisms and capabilities of an industrial programming language. (iii) The evaluation demonstrates the effectiveness of the approach in an industrial setting by computing the inactive code of different configurations. Results of the evaluation in Section 4.1.4 show that the approach provides accurate results, which were verified with a domain expert.

The second part (Section 4.2) then relies on the method for inactive code detection and presents a technique for the recovery of feature-to-code mappings. It contributes a novel approach exploiting the synergies between program analysis and diffing techniques to reveal feature-to-code mappings for configurable software systems. The evaluation in Section 4.2.1 demonstrates the feasibility of the approach with a set of KePlast products.

Section 4.3 discusses related work and Section 4.4 concludes this chapter with a summary.

4.1 Identification of Inactive Code

4.1.1 Problem Illustration

This section illustrates how inactive code is detected and why it is difficult to handle it manually.

Inactive code is defined as the code that is contained in the source code of a concrete product variant but may never be executed. This can happen, e.g., when configuration options are used to enable functionality in the program and when the options' values are bound at load time. Inactive code arising from load-time configuration options cannot be detected by a compiler which cannot know their value at the time of compilation.

Listing 4.1 illustrates the problem with a configurable program. Using the class `java.util.Properties` to load the value of configuration options `transactions` and `logging`. The value of configuration option `transactions` is used in method `main` in the branch statement. The consequent branch of this statement contains calls to the methods `start`, `commit`, and `doOperation`. In the alternative branch, the method `doOperation` is called a second time. There are also two nested branch statements in lines 10 and 19 testing if the logging functionality is enabled. The if statement testing the configuration option's value is called the *activator code*, deciding about the execution of the implemented functionality.

```
class Main {
  static Properties p = Properties.load("conf.prop")

  public static void main(String[] args) {
    if("on".equals(p.getProperty("transactions"))) {
      Transaction t = new Transaction();
      t.start();
      doOperation();
      t.commit();
      if("on".equals(p.getProperty("logging"))) {
        log("transaction successful");
      }
    } else {
      doOperation();
    }
  }
  public static void doOperation() {
    // ...
    if("on".equals(p.getProperty("logging"))) {
      log("operation completed");
    }
  }
  public static void log(String msg) { /* ... */ }
}

public class Transaction {
  void start() { /* ... */ }
  void commit() { /* ... */ }
}
```

Listing 4.1: Small configurable program.

Assume that the configuration option `transactions` is not enabled and therefore the calls to methods `start`, `doOperation`, and `commit` will not be executed. However, method `doOperation` is called a second time in the alternative branch. Further assume that configuration option `logging` is enabled. At first sight, it also seems that method `log` is inactive since it is used in an inactive branch. However, method `doOperation` also uses the logging functionality.

In this illustration, it is very easy to find out that the methods `start` and `commit` are inactive while the methods `doOperation` and `log` are still executed. In complex systems, the problem is of course not that obvious. As observed in the industry partner's source code, there are many places for code activations and deep call chains with many shared methods. Doing such an analysis manually would be cumbersome and error prone.

4.1.2 Approach

Figure 4.1 illustrates the ICD approach, which used the CSDG introduced in Chapter 3 as the basis for identifying the inactive code in a given product variant. First, the

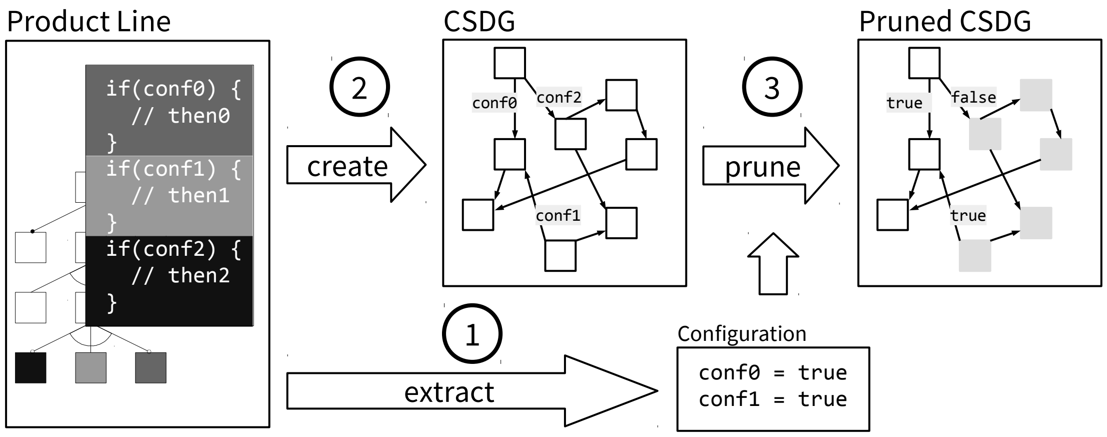


Figure 4.1: The approach identifies inactive source code in product variants by computing a CSDG and evaluating presence conditions based on a given system configuration.

concrete product configuration is transformed into assignments to the variables used in the presence conditions. Therefore, this transformation is a complementary part of the second step of building the CSDG of extracting the presence conditions and establishes a semantic link between a configuration and the statements in the program.

Identifying inactive code is then accomplished by a reachability analysis in the CSDG. Initially all nodes in the CSDG are marked inactive. Then our algorithm starts at the `root` procedure node. The reachability analysis recursively follows the edges and marks all reached nodes active until all reachable nodes are visited. If a presence condition is attached to an edge, it is followed only if the condition is satisfied regarding the values of the variables representing the product configuration. As a final step, the information from the active and inactive nodes is transferred to the corresponding statement nodes in the AST.

Semantics

Formally the inactive code detection problem can be defined as follows. Recall the definition of the CSDG (cf. Section 3.5), i.e.,

$$CSDG = (V, E, T, C, PC)$$

Assume

$$v_0 \in V$$

to be the start node representing the single entry point of the program. Let

$$cvals \in \{True, False\}^C$$

be an assignment of Boolean values to configuration variables and therefore represents a concrete configuration. Then, a pruned SDG

$$SDG_p = (V_p, E_p, T_p)$$

representing the SDG of the active code for the product configuration *cvals* is constructed as follows. Let $(v_0, v_1, v_2, ..., v_n) \in V^*$ be a valid path from node v_0 to node v_n, i.e., $v_i \xrightarrow{control} v_{i+1} \in E$, $i \in \{0, 1, 2, ..., n-1\}$. The set of nodes $V_p \subseteq V$ is the set of nodes v_i reachable from start node v_0 along any valid path $(v_0, v_1, ..., v_n)$ where all the edges $v_i \xrightarrow{control} v_{i+1}$ are control dependence edges and all the presence conditions for edges evaluate to true for the current configuration. Formally

$$V_p = \{v_n \in V | \exists (v_0, v_1, ..., v_n) \in V^* :$$
$$\forall i \in \{0, 1, 2, ..., n-1\} : v_i \xrightarrow{control} v_{i+1} \in E \wedge PC(v_i \xrightarrow{control} v_{i+1}, cvals)\}$$

Then, the set of edges $E_p \subseteq E$ is the remaining set of edges between nodes in V_p, i.e.,

$$E_p = \{v_1 \rightarrow v_2 | v_1 \rightarrow v_2 \in E \wedge v_1, v_2 \in V_p\}$$

The function T_p is just the function T restricted to the domain E_p, i.e.,

$$T_p : E_p \rightarrow \{control, data\}$$

with $T_p(e) = T(e)$. Note that the result of pruning is a SDG and not a CSDG, i.e., the variability information is abandoned.

Algorithm

Algorithm 1 shows the algorithm for identifying inactive code in pseudo code. It takes three parameters, (i) the CSDG, (ii) the node representing the single entry method of the system, and (iii) the concrete product configuration. The CSDG is the data structure $CSDG = (V, E, T, C, PC)$ (cf. Section 3.5), the second parameter is a method node $v_0 \in V$ and the third parameter is a Boolean vector $cvals \in \{True, False\}^C$. The algorithm performs a reachability analysis on the CSDG only following the control dependencies if their associated presence conditions are satisfied. Function $PC(csdg, e, conf)$ selects and applies the CSDG's presence condition function. The result of the algorithm is then a set of unvisited nodes which are not reachable in the control dependence subgraph meaning that they cannot be executed with the provided configuration.

Figure 4.2 further illustrates how Algorithm 1 works. The left side shows the initial control dependence subgraph of the program in Listing 4.1. Initially, all nodes are

Algorithm 1 The algorithm for identifying the inactive code.

```
procedure IdentifyInactiveCode($csdg = (V, E, T, PC)$, $v_0$, $cvals$)
    $visited \leftarrow \emptyset$
    $queue \leftarrow \{v_0\}$
    while $queue \neq \emptyset$ do
        $n \leftarrow$ removeFirst($queue$)
        $visited \leftarrow visited \cup \{n\}$
        for $e = (n \xrightarrow{control} s) \in E$ do
            if $s \notin visited \land s \notin queue$ then
                if $PC(e, cvals)$ then
                    $queue \leftarrow queue \cup \{s\}$
                end if
            end if
        end for
    end while
    return $V(csdg) \setminus visited$
end procedure
```

unvisited and therefore gray. Starting at node `main`, the algorithm follows each outbound control dependency and marks every visited node. If there is a presence condition at an edge, the formula is evaluated by applying the provided configuration. Furthermore, if the presence condition evaluates to $false$, the edge is ignored and the successor will not be visited over this edge. The right side of the figure shows the result of this example assuming configuration option *transactions* is disabled and *logging* is enabled, i.e., $trans = false \land log = true$.

4.1.3 Implementation

The approach has been customized and implemented for the KePlast product line (refer to Section 1.2) of the industry partner KEBA AG to demonstrate its feasibility and flexibility. To do so, the analysis framework presented in Section 3.6, which specifically targets the programming language of the industry partner, builds the CSDG required for this approach. The analysis framework needs to know the used variability patterns to extract seed conditions. Therefore, first it is described what kind of patterns occur in KePlast programs. Second, it is then described how configuration files are used as a concrete product configuration to evaluate the presence conditions.

Used Variability Mechanisms

In order to implement variability, KEBA uses three mechanisms that have been identified based on expert knowledge and manual review of the source code. All three mechanisms

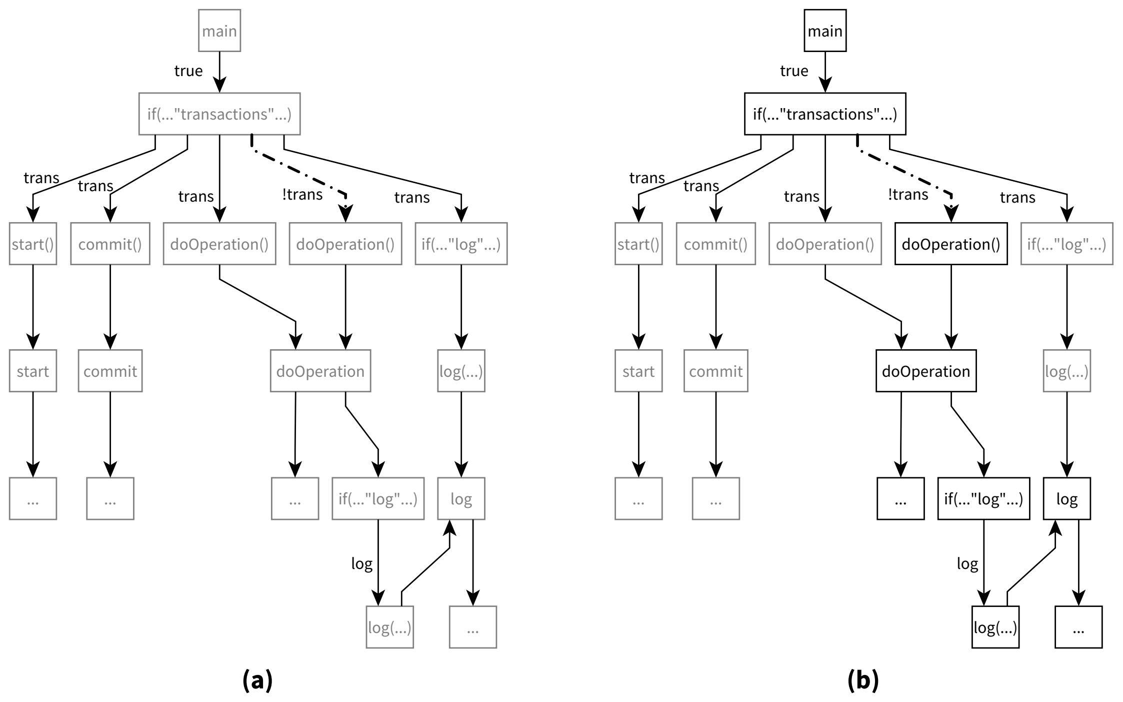


Figure 4.2: Illustration of the algorithm for identifying inactive code. **(a)** shows the control dependence subgraph before pruning and **(b)** shows the result.

belong to the class of load-time variability, which is bound at program startup. The three mechanisms are as follows:

System variables are special program variables representing endpoints to hardware machine equipment like sensors and actuators. Therefore, these variables are declared globally in the program. They are also an important variability mechanism to represent optional equipment used in a product configuration. For example, the system variable `di_ImpulseInput` represents an optional digital input sensor for providing impulses of a rotating machine component. For testing the presence of system variables in the program as depicted in Listing 4.2, a function `IS_LINKED(var)` is available and heavily used in KePlast for implementing variation points.

```
IF IS_LINKED(di_ImpulseInput) THEN
  // ...
END_IF
```

Listing 4.2: Code snippet testing if a system variable is available.

Configuration parameters are another widely used variability mechanism. The KePlast user interface provides special configuration forms for setting configuration parameters prior to system start-up. For example, the parameter `buzzer_signal_duration` defines

```
IF buzzer_signal_duration > 0 THEN
// ...
END_IF
```

Listing 4.3: Code snippet testing if some functionality is enabled.

```
// Hydraulic Machine
PLUG_CONNECTION
  fb0 : FB_HYDRAULIC;
END_PLUG_CONNECTION

// Electric Machine
PLUG_CONNECTION
  fb0 : FB_ELECTRIC;
END_PLUG_CONNECTION
```

Listing 4.4: Code snippet testing if some functionality is enabled.

how long an alarm will be signaled by the buzzer. As illustrated in Listing 4.3, variable `buzzer_signal_duration` can be zero deactivating this functionality.

Module linking is another mechanism. Modules represent loadable program units in the target language and are used as an elementary variability mechanism in KePlast. It is common practice to implement different variants of a control function in different modules and to decide which variant to use by specifying the respective module in a configuration file. For example, KePlast supports hydraulic machines and electric machines. Thus, for the main control components separate module variants exist supporting either the hydraulic or electric machine. To represent this specific situation, Boolean variables are introduced as presence conditions in the CSDG representing the chosen module. For example, Listing 4.4 shows how to define the two different modules for different machine types. The so-called *plug connection* allows to specify a type for a variable `fb0`. Depending on the machine type, i.e., hydraulic or electric, the implementation is specified by using the corresponding type.

Configuration Files

A prerequisite for identifying inactive code of product configurations is to create and set the variables used in the presence conditions. Besides supporting different variability implementation mechanisms, the approach further needs to be tailored to support different types and formats of configuration files used during the staged configuration process.

For instance, engineers use configuration files to define the required modules. In this case the variables representing the modules to be linked are set to true while all other module variables remain false. There are also text files containing system variable declarations which correspond to the hardware input/output ports. Listing 4.5 shows an example of a configuration file defining which input/output ports are available. For example, the identifier `IO.ONBOARD.DI:0` is the address of an input/output port and the line `name="Subsystem.di_Closed"` specifies the program variable connecting to this port. If there is an entry for a program variable in the configuration file, then the call `IS_LINKED(di_Closed)` will evaluate to true in `Subsystem0`. Again, all variables representing system variables in the product configuration are set to true while all other remain false.

Finally, the parameter settings defined in the dedicated configuration forms are written to an initialization file in the form of key-value pairs. The values are retrieved at start-up to set the variables used in the presence conditions accordingly.

```
[IO]
  [IO.ONBOARD]
    [IO.ONBOARD.DI:0]
      name="Subsystem0.di_Closed"
    [IO.ONBOARD.TI:7]
      name="Subsystem1.ti_OilTemp"
    [IO.ONBOARD.DI:18]
      name="Subsystem2.di_ImpulseInput"
```

Listing 4.5: Configuration file example defining input/output ports.

4.1.4 Evaluation

The evaluation was performed using the approach as implemented and customized for KEBA's KePlast product line. It investigates the effectiveness and accuracy of the approach for identifying inactive code. The effectiveness shows how much code can be identified as inactive for different configuration options. The accuracy evaluation compares the results of the automated analysis to the results of a manual code analysis performed by a domain expert of KEBA.

Effectiveness

It has been shown that the cost of code maintenance increases with source code size and complexity [Banker93]. Therefore, effectiveness answers to what extent the ICD approach can ease maintenance tasks for application engineers by reducing code size. In particular, the code that is identified and marked as inactive for different configuration options is measured. The metrics *inactive code size* and *inactive code scattering* are used for assessing the effectiveness of the approach.

The metric *inactive code size* is measured in lines of code (LOC), counted based on the AST which maintains traces to source code lines for that purpose. The LOC metric

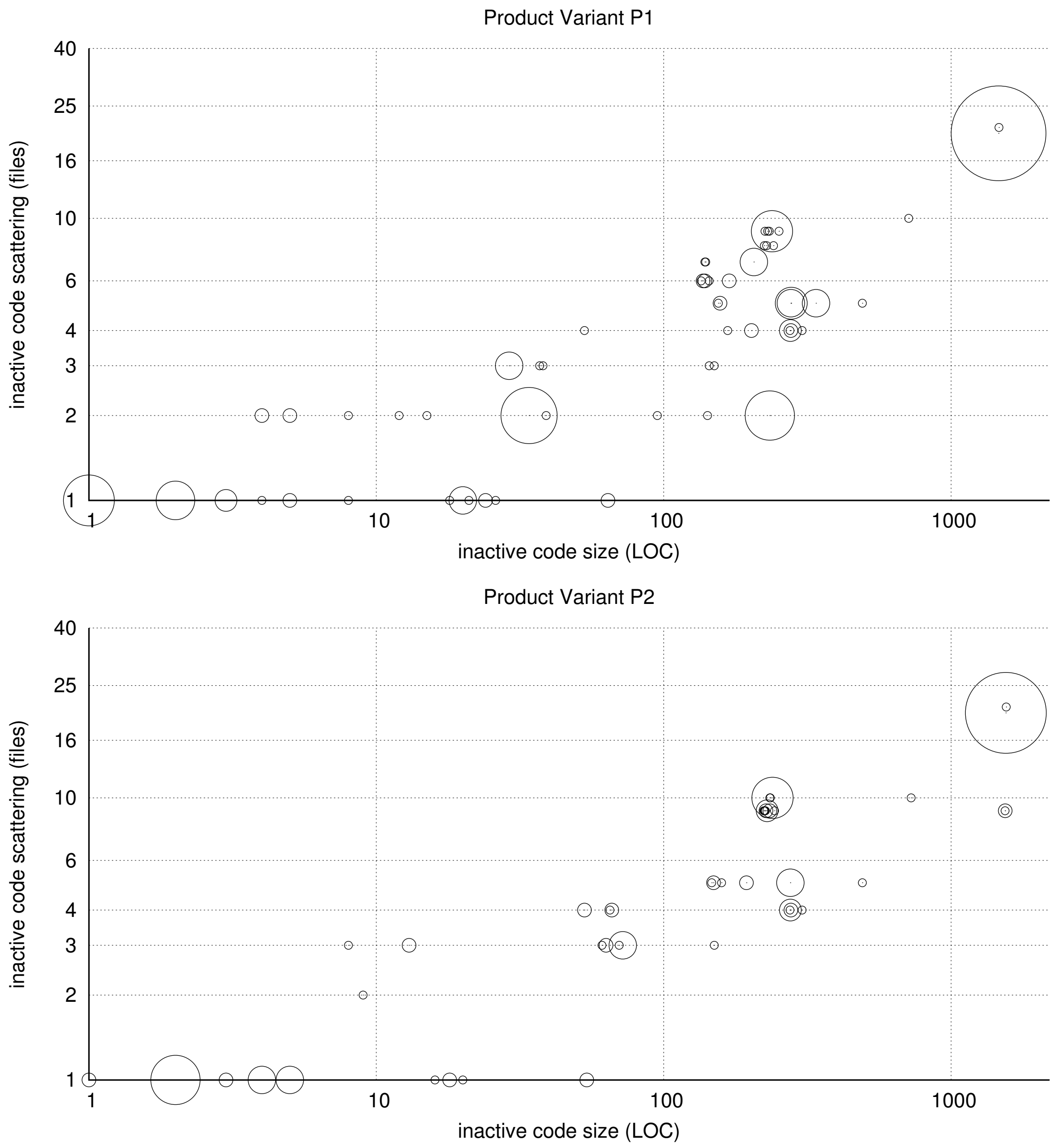


Figure 4.3: Effectiveness of the approach for individual configuration options of KePlast product variants P1 and P2. The x-axis represents the identified inactive code size (LOC). The y-axis represents the scattering of inactive code (files). For both axes a logarithmic scale is used. The size of the bubbles indicate the number of configuration options with equal data points.

considers only code lines containing statements and neglects empty lines, comment lines, and lines with closing tags (e.g. `END_IF`). Thus, our LOC metric conforms to popular code counting standards [Park92].

The metric *inactive code scattering* metric indicates to what extent the code belonging to a configuration option is spread across files. This is important from the perspective of product line maintenance as it is more difficult to understand the effects of configuration options when implementations are distributed across multiple files and components.

The evaluations are applied to the code bases of two different product variants. Product variant *P1* represents a KEBA customer project and was created by first selecting a base product using the AppCo [Lettner13] composition tool and then further customizing it to address customer requirements. The size of *P1* is 102k LOC and it has 247 configuration options. The second product variant *P2* was created using the AppCo composition tool. Therefore, the product *P2* represents a standard product configuration. The size of *P2* is 72k LOC and it has 146 configuration options. Note that product *P1* is significantly larger due to the customer-specific extensions.

For measuring the effectiveness of the ICD approach, the test program individually disabled each configuration option in *P1* and *P2* by assigning a value logically disabling the option while keeping all other configuration options unchanged. Then, the inactive code (LOC) was measured as well as the scattering of inactive code (files). Observations revealed that more than 90% of the configuration options affect only single source code modules. Therefore, the inactive code size and scattering metrics are computed for individual modules to reflect the typical work mode of developers. When a configuration option was used in several modules, it was measured several times, separately for each module using it.

Results Figure 4.3 shows the inactive code size in LOC (x-axis) and scattering in number of files (y-axis) for the different configuration options in *P1* and *P2* using a logarithmic scale. Each data point in the diagram represents a configuration option. As some configuration options have identical or almost identical data points (see next paragraph for an explanation), the diagrams use bubbles to indicate the number of configuration options with equal data points.

Discussion The results for the evaluation of effectiveness in Figure 4.3 show a significant size of inactive code for most of the configuration options. Many are in a range between 200 and 1000 LOC, some even exceed 1000 LOC. Furthermore, the inactive code also scatters over several files for a majority of the configuration options. Therefore, the inactive code will not be obvious to a developer performing a maintenance task. As expected, there is a strong correlation between inactive code size and inactive code scattering. The Spearman's rank correlation coefficient is 0.98 for product *P1* and 0.97 for product *P2*.

However, the number of data points shown in Figure 4.3 is smaller than the number of configuration options available in the two product variants (247 in *P1*; 146 in *P2*). In fact, just 53 and 44 different data points exist for *P1* and *P2*, respectively. There are three explanations for this phenomenon: first, there are many code duplicates with only small differences. Therefore, the inactive code size of two different configuration options can be the same in terms of LOC even if the inactive code lines are disjunct. Investigations revealed that the amount of code clones is significant which can be explained by the development process. Second, in some modules a lot of code is never used independently of the configuration options, e.g., because some types have never been instantiated. In particular, this happens in library modules implementing basic data structures and algorithms. Third, disabling specific configuration options can result in activating code while other code becomes inactive and vice versa.

Accuracy

To perform the evaluation for accuracy, a domain expert of our industry partner manually analyzed the inactive source code for selected configuration options and compared the results of the manual analysis with the results from the analysis tool. The domain expert is the development lead of the investigated SPL and has in-depth knowledge about the entire KePlast system. Due to time constraints of the expert, it was necessary to focus on one product variant, i.e., product variant *P1*, and a limited set of configurations options covering both small and large inactive code and different scattering. Table 4.1 shows the five configuration options together with the inactive code size (IC) and inactive code scattering (IS) computed by the tool. Note that these configuration options do not necessarily have a Boolean type. They can be logically disabled by removing an entry from an external configuration file meaning that the configuration option will then never be defined.

The evaluation comprised three steps:

(1) The domain expert was briefed about the goals of the analysis and the task.

(2) The domain expert used KEBA's IDE to mark the source code that becomes inactive as a result of disabling the selected configuration option. The expert first performed a text-based search for the variable implementing the configuration option currently explored. All locations found were then further investigated. Furthermore, the expert explored all dependencies (e.g., value assignments to a global variable, procedure calls, etc.). Code considered as inactive was commented out in the IDE. It took the domain expert 135 minutes in total to investigate the five configuration options.

(3) A researcher compared the output of the manual code marking with the automated

Option	IC [LOC]	IS [#files]	RC [LOC]	TC [LOC]	Acc
di_ImpulseInput	53	4	13	0	80.3%
ti_OilTemp	64	1	9	0	87.7%
di_EmergencyStop	278	5	2	275	99.3%
di_ButtonManual	339	5	1	275	99.7%
ao_Valve	712	10	33	215	95.6%

Table 4.1: Inactive code size (IC) and inactive code scattering (IS) for the investigated configuration options. RC denotes the number of lines only found by the domain expert. TC denotes the number of lines only found by the tool. ACC is the accuracy of the computed results compared to the manual identification.

analyses results. A line-based diffing tool was used to compare each modified copy of the source code to the unmodified source code. Three cases were distinguished: (i) the domain expert's assessment and the tool results match, (ii) the domain expert identified inactive code not found by the tool, (iii) the tool identified inactive code not found by the domain expert. In cases (ii) and (iii) the author reviewed the conflicting statements to find the reason for the divergence.

Results The results of the comparison and the accuracy are depicted in Table 4.1. Column RC shows the reported conflicts in LOC corresponding to conflict case (ii), i.e., the numbers denote the LOC that have not been marked by the tool but by the domain expert. Column TC shows the reported conflicts in LOC corresponding to conflict case (iii), i.e., the numbers denote the LOC that have not been marked by the domain expert but by the tool. The accuracy is based on this number and computes by the fraction of inactive code found by the tool related to the inactive code found by the domain expert: $ACC = \frac{IC}{IC+RC}$.

Discussion The conclusions from our accuracy evaluation are twofold: on the one hand, manually identifying inactive code is a tedious and time-consuming task. Although the only objective was to identify inactive code per configuration option, it took on average 27 minutes. This is significant given that typical products have more than 100 configuration options.

Table 4.1 shows that the domain expert identified some inactive code which was not found by the tool (column *RC*). The domain expert also removed declarations when the declared element was not used anymore. This is not supported by our tool currently, however, the tool could easily be extended with a feature for finding unused variables after the identification of inactive code. Furthermore, the domain expert removed statements which were active but did not have any effect, e.g., assignments to variables which are never read. Again, extending our approach to find such irrelevant statements would be straightforward.

On the other hand, for some configuration options the tool found a lot of inactive code which was not identified by the domain expert (cf. column *TC* in Table 4.1), thus confirming the importance of our automated approach. In these cases the inactive code found by the tool is not directly related to the configuration option and was thus not marked by the domain expert.

Threats to validity The evaluation of the approach is based on a single product line in a specific application domain and for a specific programming language. Our evaluation did not demonstrate how well the approach would work for other programming languages and PLs. However, there is a high confidence that the approach works well in other settings. A prerequisite is that the variation points can be evaluated at load-time based on configuration options, which is a common technique used in PLs. Reisner et al. [Reisner10], for example, show that open source systems heavily use configuration options to support creating different software variants. A further prerequisite for our approach is that it must be possible to build an SDG.

Effectiveness was evaluated only for two partially preconfigured product variants of KePlast. However, P1 and P2 are representative examples of product variants created in a staged configuration process based on the KePlast system. The evaluation showed good results for both variants and since the architecture of the system is the same for other product variants, results will be similar.

Due to limited availability and time constraints, accuracy was determined with a single domain expert from industry and a small set of configuration options. However, the results from the automated analysis matched for the configuration options investigated by domain expert. An exception was active but irrelevant code which was also identified by the domain expert but is not considered in our current implementation of the approach.

4.2 Automated Extraction of Feature-to-Code Mappings

The second part of this chapter relies on the ICD approach and presents a technique for the recovery of feature-to-code mappings. As pointed out in Chapter 1 software product lines are rarely planned and developed from scratch. Instead they are typically the result of maintaining and evolving code bases over many years by using, for example, clone-and-own approaches. However, existing techniques supporting the development and maintenance of SPLs frequently rely on feature-to-code mappings to relate high-level variability abstractions such as features to *variation points*— the locations in artifacts where variability occurs. Due to the unplanned nature of such systems, feature-to-code mappings either do not exist or are frequently outdated.

Linsbauer et al. [Linsbauer13] and Fischer et al. [Fischer14] present the extraction and composition for clown-and-own (ECCO) approach that allows recording variability traces in clone-and-own product lines. ECCO compares different program variants to extract feature-to-code traces, interactions between features, and dependencies between traces. It assumes as input a set of n product variants about which two things are known:

(i) the source code that implements each product variant and
(ii) the set of features that each program variant provides.

It is not important how these product variants came to be. They can be separately maintained variants created by a manual clone-and-own approach or product variants generated using a more structured approach like preprocessor annotated source code.

The output of ECCO are traces defining how source code is related to features. For instance, a trace defines for a particular part of the code which features it implements likely, at most, or certainly not [Fischer14]. ECCO also considers *feature interactions*, i.e., traces that refer to source code that is present only when all interacting features are present, and *negative features*, i.e., traces that refer to source code that is included in case of the absence of a feature. ECCO therefore uses the concept of modules similar to Liu et al. [Liu06]. Such traces are represented as presence conditions called modules similar to Liu et al. [Liu06], of the form $\delta^i(\overline{f_0}, \overline{f_1}, ..., \overline{f_n})$ where n is the order of interaction, i.e., the number of interacting feature pairs. The list contains positive or negative features $\overline{f_i}$ where $\overline{f_i} \in \{f_0, \neg f_0, f_1, \neg f_1, f_2, \neg f_2, ..., f_n, \neg f_n\}$, i.e., f_i means the presence condition requires feature f_i to be enabled and $\neg f_i$ means the presence condition requires feature f_i to be disabled. If the module contains just one feature, i.e., $\delta^0(f_0)$, then this denotes the bare feature. If the module contains two features, e.g., $\delta^1(f_0, f_1)$, then this denotes the interaction of two features and the module is only present if the features are present at the same time. Furthermore, ECCO also extracts dependencies between traces. For example, a trace A that contains a statement calling a method which is part of another trace B depends on that trace B. Based on this information ECCO constructs a variability model that can be used as a starting point for manually defining a feature model.

Lastly, ECCO also comes with a composition tool that can use the previously extracted information to compose product variants with given sets of features they shall implement.

ECCO identifies the feature traces based on the differences between product variants. The results of this analysis can be used to integrate different product clones into a single system representing the variability mined in the product variants. Thus, the ECCO approach relies on differences in products' source code which show the presence of feature combinations. However, when using load-time configuration mechanisms, ECCO fails as no differences can be observed in the source code. Figure 4.4 shows an example of two product variants. In order to compute the feature traces, ECCO compares the two product variants with respect to structural differences. However, there are no structural differences since the program using load-time configuration options and the source code of all features is always present. Therefore, the ICD approach and the ECCO approach are combined for identifying feature-to-code mappings in such systems. First ICD identifies the active and inactive code for a specific product configuration. ECCO then computes a feature-to-code mapping which not only considers which code is present but also which code is finally active in a product configuration. Figure 4.4 illustrates this process. The listing on the left shows the program with feature *transactions* enabled and the listing on the right shows the same program with the feature disabled. The lines with gray background indicate the inactive code. Without having the information of inactive code, ECCO cannot determine any structural differences between the variants except of a very small difference in the configuration file. Using inactive code detection, ECCO is able to work with load-time configuration options properly. In this way, the integration of program analysis and diffing techniques improves the automated recovery of feature-to-code mappings.

In the following, the combination of the ICD and ECCO approach will be shown based on a case study.

4.2.1 Evaluation

The goal of this evaluation is to show that extracted traces and the dependencies among the traces are consistent with the feature model. For this purpose, we applied the combination of the two approaches to the industry partners software system KePlast.

Recall, the industry partner KEBA uses a multi-stage product configuration process (cf. Section 1.2). In the first stage, the AppCo configuration tool selects components for inclusion in the final product variant on a coarse-grained level. However, the components are still configurable and they use load-time configuration options for this task. This configuration process causes the problem of inactive code such that the actual code of a product variant can only be determined when considering the load-time configuration options.

Since the evaluation requires a feature model, we reverse-engineered a feature model for the Mold1 component of KePlast. Feature models have been developed to describe the variability of selected subsystems. The feature model for component Mold1 was reverse-engineered in two stages. First we analyzed KEBA's custom-developed configuration tool, which is used for deriving a base system solution by selecting components and defining initial configuration settings. Our analysis resulted in an initial feature

```
class Main {
 public static void
 main(String[] args) {
  if(..transactions...) {
   Transaction t =
     new Transaction();
   t.start();
   doOperation();
   t.commit();
   if(...logging...) {
    log(...);
   }
  } else {
   doOperation();
  }
 }
}

public class Transaction {
  void start() {
    ...
  }
  void commit() {
    ...
  }
}
```

(a) $\delta^2(base, transactions, logging)$

```
class Main {
 public static void
 main(String[] args) {
  if(..transactions...) {
   Transaction t =
     new Transaction();
   t.start();
   doOperation();
   t.commit();
   if(...logging...) {
    log(...);
   }
  } else {
   doOperation();
  }
 }
}

public class Transaction {
  void start() {
    ...
  }
  void commit() {
    ...
  }
}
```

(b) $\delta^1(base, logging)$

Figure 4.4: Two product variants of a small SPL with one feature.
(a) Product variant with feature *transitions* enabled.
(b) Product variant having feature *transitions* disabled.

model for the component Mold1. Second we discussed and refined this initial feature model with the development lead of the KePlast platform. The feature model is shown in Figure 4.5a.

In order to fulfill the goals, the evaluation investigates two research questions RQ1 and RQ2.

RQ1. Does the extracted trace information correctly reflect the variability in the system?

To answer this question we use ECCO's compositor which uses the extracted trace information to generate product variants. We use it to re-compose the same products that were used as input (i.e., the products with the same features) based solely on the extracted trace information. This means that we decide about including a trace's code based on the features of that product variant. We then compare these re-composed product variants to the corresponding original product variants generated by the KePlast

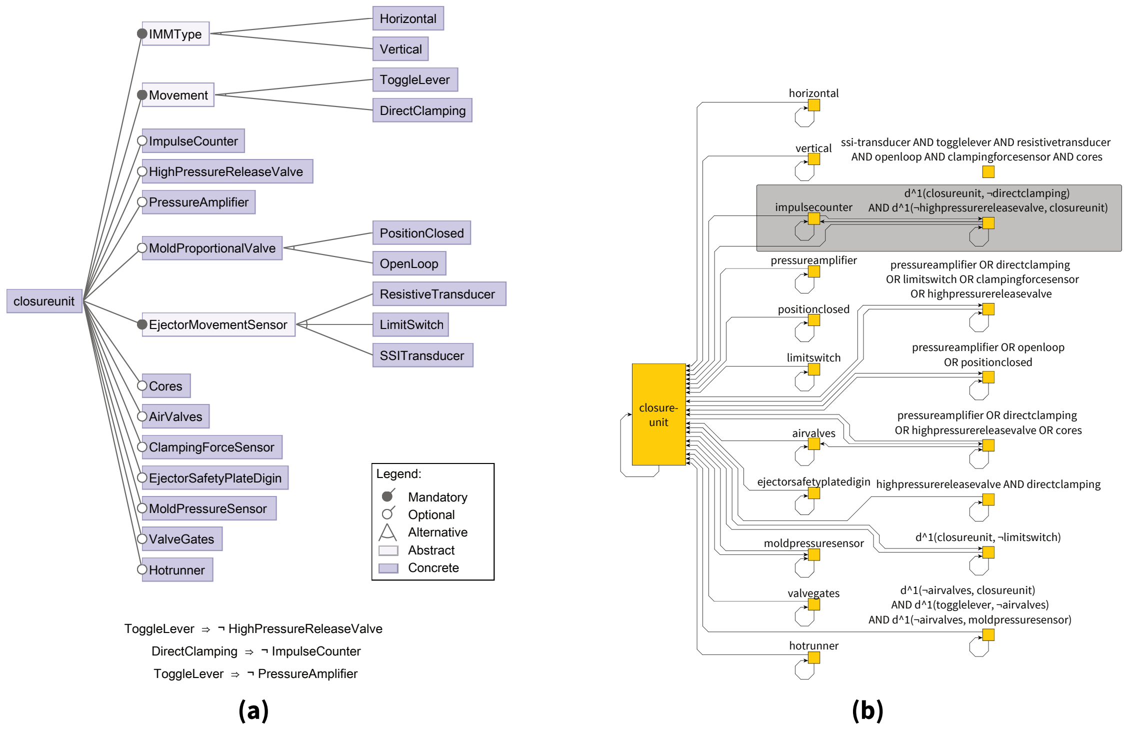

closureunit
IMMType
Horizontal
Vertical
Movement
ToggleLever
DirectClamping
ImpulseCounter
HighPressureReleaseValve
PressureAmplifier
MoldProportionalValve
PositionClosed
OpenLoop
EjectorMovementSensor
ResistiveTransducer
LimitSwitch
SSITransducer
Cores
AirValves
ClampingForceSensor
EjectorSafetyPlateDigin
MoldPressureSensor
ValveGates
Hotrunner
Legend:
Mandatory
Optional
Alternative
Abstract
Concrete
ToggleLever ⇒ ¬ HighPressureReleaseValve
DirectClamping ⇒ ¬ ImpulseCounter
ToggleLever ⇒ ¬ PressureAmplifier
(a)
closure-unit
horizontal
vertical
ssi-transducer AND togglelever AND resistivetransducer AND openloop AND clampingforcesensor AND cores
impulsecounter
d^1(closureunit, ¬directclamping) AND d^1(¬highpressurereleasevalve, closureunit)
pressureamplifier
pressureamplifier OR directclamping OR limitswitch OR clampingforcesensor OR highpressurereleasevalve
positionclosed
limitswitch
pressureamplifier OR openloop OR positionclosed
airvalves
pressureamplifier OR directclamping OR highpressurereleasevalve OR cores
ejectorsafetyplatedigin
highpressurereleasevalve AND directclamping
moldpressuresensor
d^1(closureunit, ¬limitswitch)
valvegates
d^1(¬airvalves, closureunit) AND d^1(togglelever, ¬airvalves) AND d^1(¬airvalves, moldpressuresensor)
hotrunner
(b)

Figure 4.5: The comparison of the feature model and the trace dependency graph.
(a) Mold1 feature model.
(b) Extracted trace dependency graph. Each node represents a trace to the code of a feature or feature interaction.

product configuration tool.

More specifically, we create *n* KePlast product variants using the KePlast configurator and use them as input to our approach. The size of the KePlast products range from 53,000 to 58,000 LOC when ignoring empty lines and comments. We then use the ECCO's compositor to re-compose these variants based on the extracted trace information. Finally, we compare the re-composed variants to the corresponding original variants generated by the KePlast configurator (after using ICD to remove the inactive code) and determine its correctness. Ideally they will all match their counterparts.

We measure the correctness of a composed product variant with respect to a ground-truth product variant generated from the KePlast configurator by means of the precision and recall metrics computed for the source code of the product variants. For example, a recall of 1.0 means that the composed product contains all the source code the original counterpart contains, and a precision of 1.0 means that the composed product contains no source code represented as AST nodes, that is not also contained in the original product variant.

Results When using $n = 15$ available KePlast product variants as input to our approach and then recomposing them using the extracted trace information, the resulting average precision is 1.0, i.e., the composed products contain no source code that is not also contained in their respective original product variants. The average recall is 1.0 which means that the composed products contain all source code of their original counterparts. This shows that the extracted traceability information is sufficient to re-compose the original product variants indicating that the extracted traces are correct.

RQ2. Do the dependencies between the traces—the implementation variability—adhere to the design variability?

To answer this question we compare the extracted trace dependency graph (i.e., the dependencies between the traces) to the feature model of the KePlast system and check whether the set of product variants described by the feature model is a subset of the set described by the extracted dependency graph (i.e., allowed by the implementation). If the feature model would allow for the creation of product variants that are not supported by the implementation then possible reasons can be that i) the extracted traces and/or their dependencies are erroneous, ii) the previously reverse-engineered feature model is erroneous, or iii) we uncovered a flaw in the KePlast software system where it is possible to create variants that will not work. Specifically, we created n KePlast product variants and used them as input to our approach. We then compared the resulting dependency graph to the manually reverse-engineered feature model.

Results The extracted trace dependency graph is shown in Figure 4.5b. The nodes are labeled with single features and feature interactions (written as δ^i(*feature_1, . . . , feature_i*) with i representing the number of interacting features). Each node represents a trace to the code of a feature or feature interaction. Negative features ($\neg$*feature*) express that the features must not be present for the traced code to be included in a product variant. The dependency graph's structure is very simple. There are almost no dependencies between traces. Dependencies mostly occur within traces or with the *base*, as expected. One trace (in the top right) does not have any dependencies. This is because there were differences in the list of features but the corresponding features could not be associated with any code.

When comparing this dependency graph to the feature model shown in Figure 4.5b one can see high similarity. To a large extent the dependency graph matches the feature model and the feature model violates only few dependencies in the graph. For example *ClosureUnit* depends on *MoldPressureSensor*. This would make the latter a mandatory feature which it should not be according to the feature model. Identifying the causes for these deviations will be part of future work. For instance, we want to determine whether there is a flaw in the KePlast system, the reverse-engineered feature model, or the extracted traces. We hope that our approach will enable software engineers to efficiently and effectively identify and reconcile differences between how variability is modeled and how it is actually realized. Additionally one can see that the feature model's cross-tree constraint *DirectClamping* ⇒ ¬*ImpulseCounter*, saying that these features exclude each other, is also reflected in the dependency graph (see highlighted area in Figure 4.5b). The trace for *ImpulseCounter* requires code elements from the trace containing ¬*DirectClamping* meaning that *ImpulseCounter* cannot be present in a product variant if also feature *DirectClamping* is present.

4.3 Related Work

4.3.1 Identifying Inactive Code

Several authors have presented analysis techniques considering program level variability and configuration mechanisms. Variability-aware program analysis techniques target to consider variability during program analysis to reduce the total analysis effort compared to analyzing every possible product variant separately. Examples are variable abstract syntax trees, variability-aware type checking, control flow graphs, and liveness analysis [Kästner11] [Liebig13]. Similarly, we also use variability information for finding inactive code. However, since in our context variability is mainly implemented using load-time configuration options instead of compile-time variability like `#IFDEF` directives, we do not need variability-aware parsing. Furthermore, in contrast to the work of Bodden et al. [Bodden13] and Liebig et al. [Liebig13] the focus is not on analyzing the code base before creating variants. We analyze already derived product variants using common analysis techniques to find source code inactive due to configuration.

Tartler et al. [Tartler09] also identifies dead code on a block level by checking the consistency between model-level and code-level feature constraints. They evaluated their approach with the Linux kernel by extracting variability constraints and dependencies of code blocks from preprocessor directives and feeding the information into a database. Furthermore, they analyzed the KConfig files and also extracted constraints and dependencies into a second database. Finally, they compared the databases to find inconsistencies such as code that will never be included, i.e., dead code. However, their work also assumes compile-time variability and the fact that code is directly annotated by the corresponding feature conditions. Furthermore, they do not follow control dependencies and can therefore not find dead code due to transitive dependencies.

Zhang and Ernst [Zhang13] present the ConfDiagnoser approach which uses static analysis, dynamic profiling, and deviation analysis to reveal the root cause of configuration errors. Their technique first performs thin slicing [Sridharan07] to determine the affected branch conditions in the source code. The program is then instrumented and the run-time behavior of these branches is recorded resulting in an execution profile. In the last step, the profile is compared to a pre-built database to detect behavior anomalies. Their approach could also find dead code due to configuration using the execution profile. However, our approach is a pure static analysis approach without instrumentation. This is in particular a requirement in our industrial setting because a developer does not necessarily have the possibility to run the code since this requires the actual hardware. Furthermore, dynamic analysis approaches are not conservative which may lead to the problem that code is identified to be dead but actually is not. This can happen if the created execution profile does not cover erroneous program runs.

Reisner et al. [Reisner10] empirically analyze how configuration options affect program behavior. In particular, the authors use symbolic evaluation to discover how run-time configuration options affect line, basic block, edge, and condition coverage for different subject programs. Furthermore, they indicate that configurable software might contain lots of inactive code, e.g., they found out that 11% of the source code of the free FTP server *vsftp* are unused in the single-processor mode, a finding that is supported by our evaluation results.

Our approach for identifying inactive code is related to program slicing [Xu05] and slicing-based change impact analysis [Arnold96] because we use the CSDG to perform our analysis. Similar to our CSDG, Snelting [Snelting96] attaches path conditions taken from conditional statements to the edges. However, our conditions represent the variability of the system only. Due to the focus of our work on variability, our analysis is able to deal with larger programs (cf. [Hammer06]). Our approach further compares to CIA techniques. According to Lehnert [Lehnert11], CIA can be used to propose software artifacts which are possibly affected by a change or to estimate the amount of work required to implement a change. CIA uses dependency analysis [Ryder01,Ren04, Badri05,Petrenko09] and program slicing techniques [Tonella03,Korpi07]. Our approach does basically the same but instead of just performing slicing, we also evaluate the presence conditions and stop slicing.

4.3.2 Recovering Feature-to-Code Mappings

Researchers have developed different automated and semi-automated approaches for recovering feature-to-code mappings. Xue et al. [Xue12] present the FL-PV approach to improve feature location in product variants by exploiting commonalities and differences of product variants. Their approach uses source code differencing formal concept analysis (FCA), and information retrieval techniques. First, they use differencing to related differences in features to differences in source code resulting in traces. Then, they use FCA to further find commonalities and differences in traces and to create minimal partitions. Third, they use latent semantic indexing (LSI) reestablish the connection between features and code elements. Rubin et al. [Rubin12] suggest heuristics for improving the accuracy of feature location techniques by analyzing multiple product variants.

However, despite successes in this field trace recovery remains a human-intensive activity. Indeed, researchers have pointed out that it is risky to neglect humans in the traceability loop [Hayes05] and studies exist on how humans recover such traces manually [Egyed10].

Relating high-level variability abstractions such as features or decisions to variation points—the locations in artifacts where variability occurs—is also addressed in product line engineering approaches that rely on inclusion conditions or presence conditions (e.g., [Heidenreich08,Dhungana11]). Researchers have investigated representing variability on model or program elements [Czarnecki05a,Heidenreich08]. For instance, Kästner et al. [Kästner09] present the CIDE SPL tool for analyzing and decomposing legacy code based on annotating code fragments. However, in practical settings feature-to-code mappings are often incomplete or unavailable. As Kästner et al. [Kästner14] pointed out, locating feature code cannot be fully automated and user involvement is required. In our context, finding the initial seed means defining a variation point which requires human expertise and domain knowledge. However, our approach assists developers as they only have to create a partial mapping manually which is used as a seed into the program code to automatically find active and inactive code elements for a particular feature by analyzing code dependencies.

4.4 Summary

The first part of this chapter introduced the *inactive code detection (ICD)* approach for finding code in product variants that cannot be executed because of the configuration. The inactive code is dead code in a certain product configuration. However, the code may be used in other configurations. The motivation and context of this work is given by the industry partner that uses load-time configuration options causing product variants to contain inactive code. The technique allows hiding inactive code in a product configuration to support application engineers. It is based on a CSDG representing control dependencies, data dependencies, and the variability of the system. The identification of inactive code is then accomplished by a reachability analysis of the graph together with an evaluation of presence conditions.

The approach has been implemented and evaluated for the KePlast product line. The implementation extends the analysis framework presented in Section 3.6 by the adaptation to the system-specific variability mechanisms for extracting presence conditions, by reading the system-specific configuration files, and by the implementation of the reachability analysis including presence condition evaluation.

The evaluation investigated the effectiveness and accuracy of the approach. The results show that the approach effectively identifies inactive code, even for large sizes and if scattered across multiple modules and files. The accuracy has been assessed by comparing the results to a manual code analysis performed by a domain expert.

The second part of this chapter showed how to use the ICD approach for recovering feature-to-code mappings. ECCO is a tool for recovering feature-to-code mappings in SPLs. ECCO has been shown to be able to successfully recover feature traces from a set of product configurations. However, ECCO assumes compile-time variability and relies on differences in the source code of product variants. When working with load-time variability, no difference in the source code can be observed as the differences in product variants are in the active and inactive code. Therefore, the ICD approach and the ECCO approaches were combined so that ECCO not only considers source code differences but also differences in active and inactive code. The evaluation shows that the combination produces sound results with respect to the input variants, i.e., it was possible to re-compose the input product variants from the extracted traces.

Chapter 5

Configuration-Aware Change Impact Analysis

The previous chapters introduced the CSDG and presented the ICD approach as a configuration-aware program analysis method using the CSDG. This chapter is based on [Angerer15] and introduces another method, the configuration-aware change impact analysis (CA-CIA) approach as one of the core contributions of this thesis.

Change impact analysis (CIA) is an essential program analysis technique for identifying possibly impacted source locations when modifying a program. As Section 1.2 reports, the industry partner KEBA needs to determine the impact of changes when evolving their solutions. Assessing the impact of changes during software evolution is a major effort and one of the most difficult tasks. However, traditional slicing-based CIA has no dedicated support for configurable software. Lifting the required program analyses to gain variability-aware CIA is a possible way but does not specifically support load-time variability (cf. Chapter 1). This chapter thus presents a delayed variability approach making CIA configuration-aware by using the CSDG and propagating variability information.

The remainder of this chapter is organized as follows. Section 5.1 illustrates why existing variability-aware program analysis is not sufficient for the analysis of configurable software using load-time variability. Section 5.2 explains the CA-CIA algorithm. Section 5.3 describes the implementation of the approach and optimizations in the implementation to allow handling large-scale software systems. The evaluation of the approach is presented in Section 5.4. Section 5.5 discusses advantages and disadvantages of the approach. Sections 5.6 and 5.7 conclude this chapter with related work and a summary.

5.1 Problem Illustration

CIA allows to automatically determine and systematically review the possibly impacted source code for changes. However, state-of-the-art CIA techniques [Chen01, Jász08, Bohner02, Black01] do not consider load-time variability. For example, Listing 5.1 shows an illustrative program example that can be configured by enabling or disabling the

```java
class Main {
  static Properties p = Properties.load("conf.prop")
  static boolean c0 = "on".equals(p.getProperty("c0"));
  static boolean c1 = "on".equals(p.getProperty("c1"));

  public static void main(String[] args) {
    A obj = new A();
    D d = new D();
    if(c0) {
      obj = new B();
      return;
    } else if(c1) {
      obj = new C();
    }
    int res = obj.foo(d);
    System.out.println(res);
  }
}
class A {
  int foo(D d) {
    return 2;
  }
}
class B extends A {
  int foo(D d) {
    return d.bar() * d.bar();
  }
}
class C extends A {
  int foo(D d) {
    return 2 * d.bar();
  }
}
class D {
  int bar() {
    if(!c1) return 1;
    return 0;
  }
}
```

Listing 5.1: Configurable program using load-time variability to demonstrate the effect of configuration on program analysis.

configuration options `c0` and `c1`. Existing CIA techniques do not provide information about the product variants that are affected by a change. Even this small configurable program shows that manually determining the set of affected products is difficult due to many dependencies. Assume a developer changes the return statement in line 36 and wants to know if there is an impact on the print statement in line 16. Figure 5.1 shows the corresponding SDG of the program. The return statement subject to change is represented by the node in the lower-right corner. Existing CIA techniques follow the forward edges and mark all visited nodes as possibly impacted by the intended change. This means that the nodes labeled `return d.bar()* d.bar()`, `return 2 * d.bar()`, `int res = obj.foo(d)` and `System.out.println(res)` are in the set of impacted statements. Therefore, for existing CIA techniques the print statement is considered impacted if the return statement is modified.

However, configuration options influence the control and data flow in the program and therefore need to be considered, i.e., one has to propagate configuration options and see if the impact really prevails by determining if the statements are executed and the data flows are valid under certain conditions.

Figure 5.2 illustrates how the variability information is propagated to gain this information. It shows four snapshots of the CSDG: **(a)** shows how seed condition `!c0` from node `IF c0` is propagated, **(b)** shows how seed condition `c0` from node `IF c0` is propagated, **(c)** shows how the presence condition of the change impact criterion `return 1` is used, and **(d)** shows how presence conditions at the final impacted statement are combined. The following describes the single snapshots in detail:

Snapshot (a): Figure 5.2a shows the first five steps of the illustrated propagation. First, the seed condition `!c0` arises from statement `IF c0` which tests a configuration value. Since the successor `IF c1` is in the else branch of the if statement, the condition `c0` is negated finally revealing the seed condition `!c0`. The first propagation step moved condition `!c0` over node `IF c1` and reaches the next seed condition `c1`. In this case, the two conditions are conjunctively combined resulting in `!c0 && c1`. The resulting condition is then moved iteratively to successors `int res = obj.foo(d)`, `C.foo`, `return 2`, and `D.bar`. Since these nodes are not related to configuration, the presence condition does not change.

Snapshot (b): Figure 5.2b shows the next five propagation steps. The seed condition `c0` is moved iteratively to successors `int res = obj.foo(d)`, `C.foo`, `return 2`, and `D.bar`. Again, these nodes are not related to configuration and do not modify the presence condition in any way. However, when the condition is moved over node `D.bar`, it is disjunctively combined with condition `!c0 && c1` because the node is a method node and the two incoming edges represent call edges. Therefore, the conditions are disjunctively combined because either of the calls may happen and

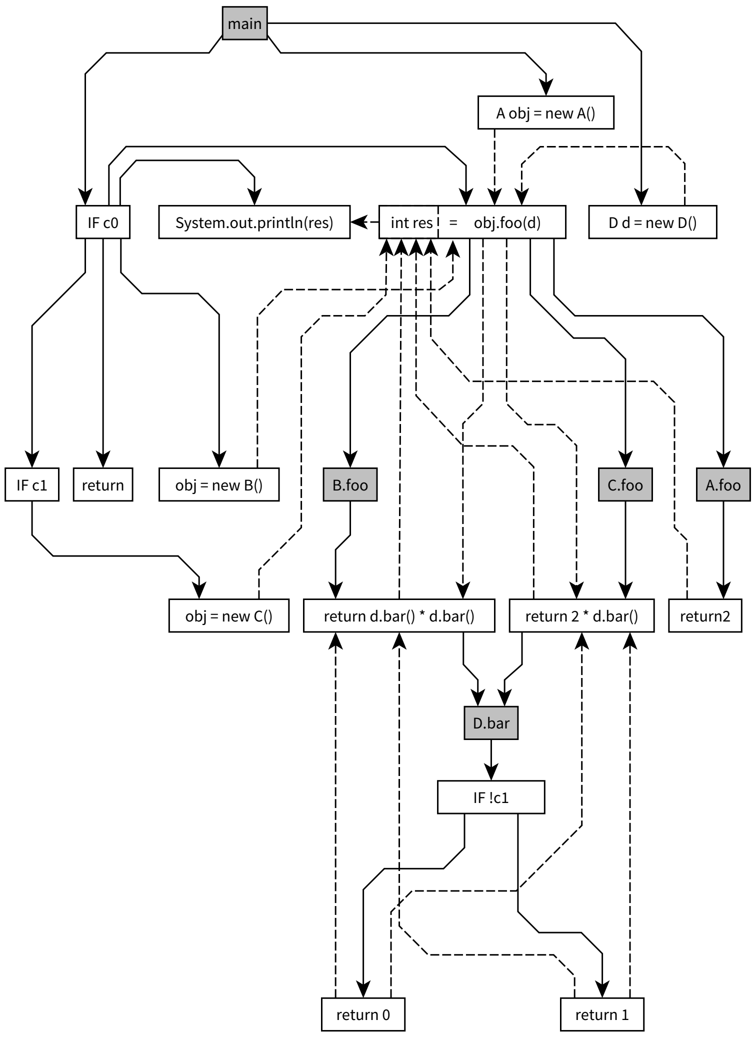
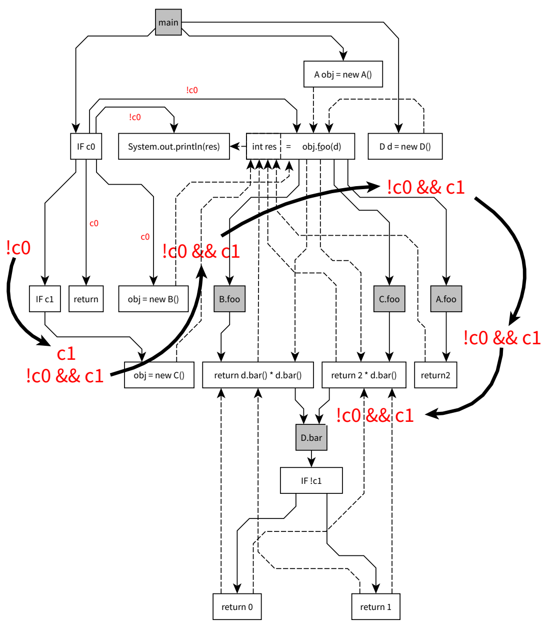


Figure 5.1: SDG for the small configurable program in Listing 5.1.

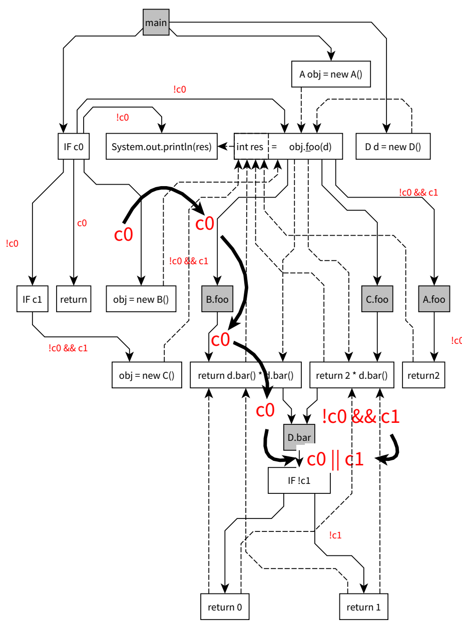
the analysis has to assume that both calls are possible. This node is thus called a join node and the two incoming presence conditions are combined to `c0 || c1`.

Snapshot (c): Figure 5.2c shows the final propagation steps and the steps for the change impact analysis. Condition `c0 || c1` from node `D.bar` first propagates over node `IF !c1` and results in condition `c0 && !c1`. Now, the presence condition is at the change impact criterion `return 1` and the change impact is computed by following the outbound edges while carrying the presence condition.

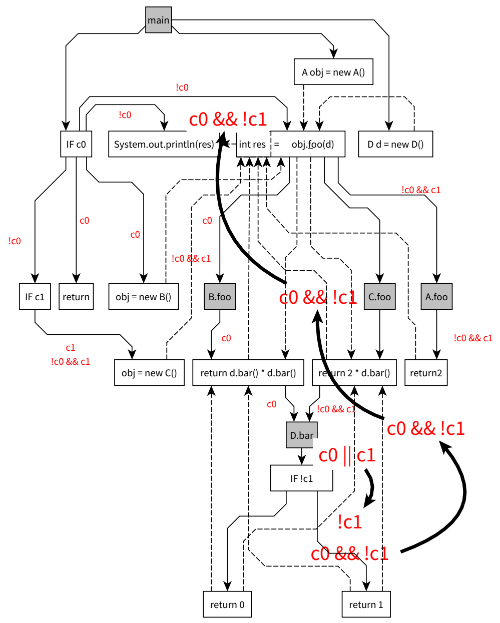
Snapshot (d): Finally, the print statement is reached (cf. Figure 5.2d) and the condition `c0 && !c1` from the change impact is combined with the incoming presence condition `!c0`. The combination is a contradiction, i.e., the result is always false and there will be no change impact to the print statement when changing `return 1`.



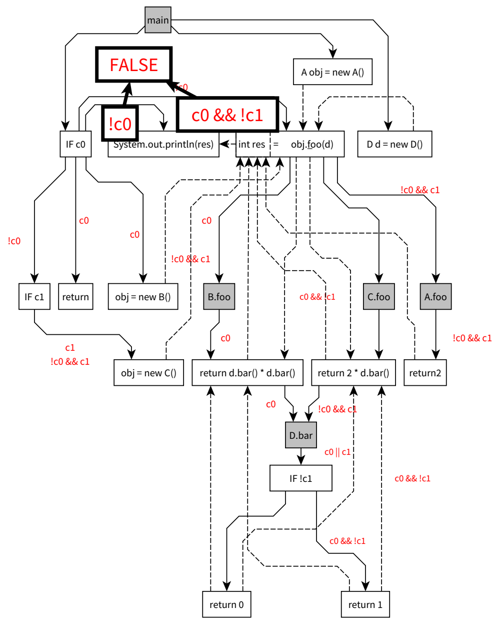
(a) Propagating seed condition `!c0` to change impact criterion.



(b) Propagating seed condition `c0` to change impact criterion.



(c) Propagating presence condition `!c0 && c1` from change impact criterion along forward dependencies.



(d) Combining presence condition of change impact criterion and of impacted statement to false.

Figure 5.2: The steps for the propagation of the variability information showing the CSDG and how the presence conditions are propagated.

```
1  class Main {
2    static Properties prop = Properties.load("conf.prop")
3
4    public static void main(String[] args) {
5      boolean c0 = "on".equals(prop.getProperty("F"));
6      boolean c1;
7      if (c0) {
8        c1 = "on".equals(prop.getProperty("X"));
9      } else {
10       c1 = "on".equals(prop.getProperty("Y"));
11     }
12
13     A obj;
14     if (c1) {
15       obj = new A1(); // indirectly depends on c0
16     } else {
17       obj = new A2(); // indirectly depends on c0
18     }
19     obj.foo();
20   }
21 }
```

Listing 5.2: Influence of configuration options on program execution.

Moreover, configuration conditions once loaded and assigned to intermediate variables may spread in the program in different forms. For example, in Listing 5.2 the value of a configuration option is stored in a local variable `c0` (line 5), which is used subsequently to decide which other configuration option to load (line 7). Currently available analysis techniques cannot handle such situations as they assume a strict separation of the variability mechanism from program control flow, i.e., they do not consider that the execution of the statements in lines 15 and 17 also depends on the configuration option `c0`. In this small example it is obvious that the execution of the statements in lines 15 and 17 also depends on conditions $c0 \wedge c1$ and $\neg c0 \wedge \neg c1$. Finally, the call in line 19 also depends on both configuration options since the reaching objects depend on both configuration options.

It is thus essential to know which configurations can reach which locations in the program. This is accomplished by a reaching definitions calculation for configuration variables. The conditions reaching a particular location in the code are therefore called *reaching conditions*.

Relating to the formal definition of the CSDG, i.e., $CSDG = (V, E, T, C, PC)$ (cf. Section 3.5), reaching conditions are defined as follows:

Definition 5. Reaching Conditions

$$RC : E \to (Var \to 2^{\{True,False\}^C})$$

where E is the set of edges in the CSDG, Var is the set of program variables, and C is a set of Boolean configuration variables.

That means RC is function mapping each edge to a function which maps variables to subsets of possible configurations. Then, possible configurations are expressed as subsets of allowed configuration vectors.

In order to represent the reaching conditions accordingly, the definition of the CSDG is extended by the reaching conditions as follows:

Definition 6. CSDG with reaching conditions (RC)
A CSDG is a tuple

$$CSDG = (V, E, T, C, PC, RC)$$

where V, E, T, C, and PC are the same as in Definition 4.

5.2 Approach

The CA-CIA approach follows the delayed variability analysis concept. It determines the configuration-aware change impact by propagating presence conditions as well as reaching conditions to CSDG nodes. A naïve approach would be to globally propagate all seed conditions until every CSDG node is annotated with the appropriate condition and then performing a forward slice starting at the statements of interest. However, it is expected that most changes will affect only a subset of the code. Propagating the conditions globally and in advance would thus be too costly. Therefore, the CA-CIA algorithm propagates presence conditions only in the required domain.

Figure 5.3 shows the steps for computing the configuration-aware change impact. The approach starts at the node for which we want to compute the change impact, i.e, the CIA criterion. First, it determines the backward slice for the CIA criterion by computing all statements that possibly have an influence on the CIA criterion. This set of nodes is the so-called *domain*. It then propagates the presence conditions and reaching conditions according to propagation rules (cf. Section 5.2.2) within the computed backward slice such that presence conditions and reaching conditions are attached to all incoming edges of the CIA criterion. Next, it determines the actual change impact by performing a forward slice, starting with the CIA criterion and propagating the presence conditions and reaching conditions.

Further, the approach needs to know the presence conditions and reaching conditions for all nodes in the change impact to compute the correct conditions. Figure 5.4 illustrates the situation where some nodes visited during the forward slice do not have presence conditions on the incoming control flow edge. The box with the dashed border denotes the domain, i.e., the set of nodes and edges that already have presence

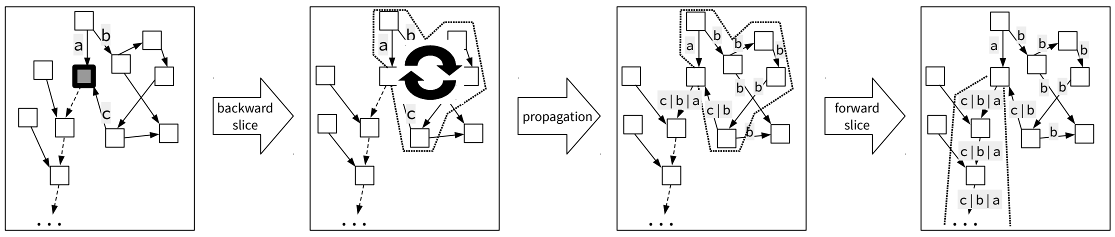


Figure 5.3: Overview of the CA-CIA approach. Starting with the CSDG containing the seeds conditions, first a backward slice is computed to determine all influencing nodes. Second, conditions are propagated within the computed backward slice. Then, the incoming condition of the CIA criterion is taken and a forward slice is computed.

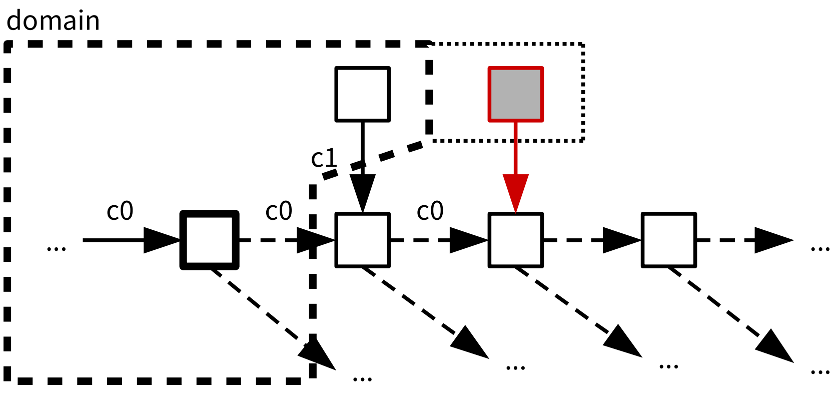


Figure 5.4: A sequence of nodes visited during a change impact analysis (forward slice). The node with the bold border is the change impact criterion. The red control edge from the red node does not have a presence condition yet.

conditions attached. The highlighted control edge from the filled node is not part of the domain and does not yet have a presence condition. However, a presence condition on this edge might also have an influence on the presence conditions of the impacted statements. Therefore, the node with this incoming edge is remembered, and in the end, a backward slice and the propagation within this slice are performed. We thus mark all nodes with incoming edges that not yet have presence conditions. After collecting these nodes, we compute a backward slice to determine the final propagation domain. Finally, we apply the propagation algorithm in this domain and label all edges in the change impact with the correct conditions. This way we do not miss any conditions and avoid global propagation across the CSDG.

5.2.1 Algorithm

Algorithm 2 shows the main algorithm for CA-CIA as outlined above. First, a backward slice starting at the CIA criterion is computed to determine the domain, i.e., the subgraph of the SDG, which has influence on the criterion. Second, the variability information, i.e, the presence conditions and reaching conditions, are propagated within this domain. In this way, the influencing variability information is brought to the CIA criterion. Next, a forward slice is performed that carries the presence condition from the CIA criterion along the dependence edges during forward slicing. The reached nodes are possibly impacted by a modification of the CIA criterion but only under the carried presence condition. Then, the nodes that have been visited but are not part of the domain, are used for a further backward slice. This is necessary because the nodes may have no presence conditions as illustrated in Figure 5.4. Once the backward slice is computed resulting in an extended domain, another propagation phase is performed to have presence conditions on all nodes.

The following algorithms Algorithm 3 to Algorithm 7 will show the individual steps.

Algorithm 2 CA-CIA algorithm

```
1: procedure ConfigurationAwareCIA($csdg = (V, E, T, C, PC, RC)$, $criterion$)
2:     $domain \leftarrow$ BackwardSlice($csdg$, $criterion$)
3:     $csdg \leftarrow$ Propagate($csdg$, $domain$)
4:     $(csdg, visited) \leftarrow$ ChangeImpact($criterion$, $domain$)
5:     $extendedDomain \leftarrow$ BackwardSlice($csdg$, $visited \backslash domain$)
6:     $csdg \leftarrow$ Propagate($csdg$, $extendedDomain$)
7: end procedure
```

Algorithm 3 defines the backward slicing algorithm as presented by Ottenstein et al. [Ottenstein84]. Since the CSDG is an extended SDG, we can simply perform a graph traversal. Therefore, we first insert the CIA criterion into the queue and then determine all reachable nodes.

Algorithm 3 The backward slicing algorithm.

```
procedure BackwardSlice($csdg = (V, E, T, C, PC, RC)$, $criterion$)
    $visited \leftarrow \emptyset$
    $queue \leftarrow criterion$
    while $queue \neq \emptyset$ do
        $n \leftarrow$ removeFirst($queue$)
        $visited \leftarrow visited \cup \{n\}$
        for $(p \rightarrow n) \in E)$ do
            if $p \notin visited$ then
                $queue \leftarrow queue \cup \{p\}$
            end if
        end for
    end while
    return $visited$
end procedure
```

Algorithm 4 is the core of the CA-CIA approach. The input parameters of the algorithm are the CSDG and the domain to perform propagation in. In particular, the domain specifies the nodes to consider while other nodes will not be visited. The algorithm starts with looking for nodes with seed conditions (cf. Section 3.3) by iterating over the nodes in the domain. The actual propagation of variability information is performed using two fringe sets. The inner loop moves conditions over nodes as long as the successor does not have a seed condition attached. If the successor edge's condition has changed, the successor node is added to the fringe. Therefore, the inner loop implements a fixed point algorithm which runs until the conditions do not change any longer. Since the algorithm stops its propagation at seeds, the conditions will always be softened and hence the inner loop always terminates. However, if a node with a seed conditions is reached, it might be necessary to also update its successors. Therefore, the successors of seed nodes are put into the back fringe to be processed in a second round.

Algorithm 4 The propagation algorithm distributing variability information within a domain.

```
procedure PROPAGATE($csdg = (V, E, T, C, PC, RC)$, $domain$)
    $frontFringe \leftarrow \emptyset$
    for $n \in domain$ do
        if hasSeed($n$) then
            $frontFringe \leftarrow frontFringe \cup \{n\}$
        end if
    end for
    repeat
        while $frontFringe \neq \emptyset$ do
            $n \leftarrow$ removeFirst($frontFringe$)
            for $e = (n \rightarrow s) \in E$ do
                if hasSeed($s$) then
                    $backFringe \leftarrow backFringe \cup \{s\}$
                else
                    $(csdg, changed) \leftarrow$ MoveCondition($sdg$, $n$, $e$)
                    if $changed$ then
                        $frontFringe \leftarrow frontFringe \cup \{s\}$
                    end if
                end if
            end for
        end while
        $frontFringe \leftrightarrow backFringe$                    ▷ exchange fringes
    until $frontFringe = \emptyset$
    return csdg
end procedure
```

Algorithm 5 is the change impact algorithm and performs a forward slice. Starting at the provided CIA criterion, it follows the outgoing edges and collects all visited nodes.

The input parameters of the algorithms are the CSDG, the change impact criterion where the analysis starts from, and the set of nodes that influence the criterion's presence condition. The algorithm performs a graph reachability analysis but also moves the presence conditions and reaching conditions during traversal. It then outputs an updated CSDG and the set of visited nodes. Note, the result is preliminary because nodes still may have control predecessors without presence conditions. This is fixed with a second propagation phase after this algorithm returned.

Algorithm 5 The condition-moving change impact algorithm formulated in pseudo code.

```
procedure ChangeImpact($csdg = (V, E, T, C, PC, RC)$, $criterion$, $domain$)
    $visited \leftarrow \emptyset$
    $queue \leftarrow criterion$
    while $queue \neq \emptyset$ do
        $n \leftarrow$ removeFirst($queue$)
        $visited \leftarrow visited \cup \{n\}$
        for $e = (n \rightarrow s) \in E$ do
            $(csdg, \_) \leftarrow$ MoveCondition($csdg$, $n$, $e$)
            if $s \notin visited$ then
                $queue \leftarrow queue \cup \{s\}$
            end if
        end for
    end while
    return $(csdg, visited)$
end procedure
```

Algorithm 6 is responsible for performing a single propagation step. It therefore has three parameters: the CSDG, the currently visited node *n*, and the target edge *e*. The function computes the incoming presence condition and reaching conditions from the predecessor edges. Algorithm 7 accomplishes this combination. Then it propagates the incoming presence condition and reaching conditions to the outgoing edge *e*. Before the successor edge is updated using `setCondition`, it checks if any condition has changed. The result of the comparison is the updated CSDG and a flag indicating a change.

Algorithm 6 Computes the incoming condition for a node, tests if the condition has changed, and returns the updated CSDG.

```
procedure MoveCondition($csdg = (V, E, T, C, PC, RC)$, $n$, $e$)
    $(pc, rc) \leftarrow$ ComputeIncomingCondition($csdg$, $n$, $e$)
    $changed \leftarrow PC(e) \neq pc \vee RC(e) \neq rc$
    $csdg \leftarrow$ setCondition($csdg$, $e$, $(pc, rc)$)
    return $(csdg, changed)$
end procedure
```

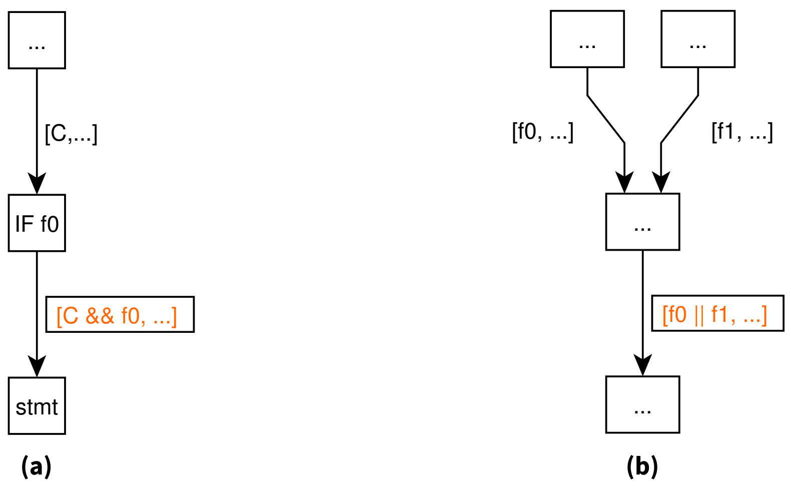


Figure 5.5: Basic cases of propagating presence conditions.
(a) Introducing a presence condition on a control edge.
(b) Joining presence conditions.

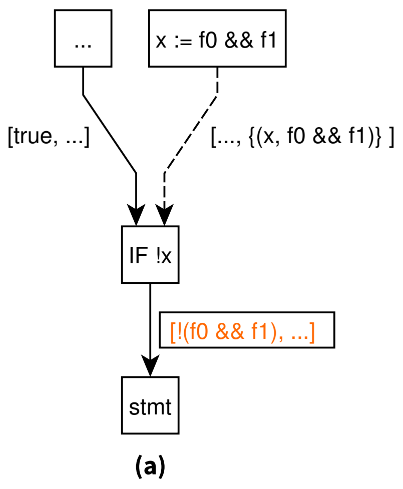
5.2.2 Propagation Cases

This section describes the basic propagation cases as performed by Algorithm 7. These propagation cases depend on the concrete type of analysis to perform and therefore the following cases are for CA-CIA.

Each case is illustrated by a fraction of the CSDG with three node levels. The upper node level are the predecessors, the middle node level contains the current node to process, and the lower level contains the successors. The node types are not specified explicitly since they results from the content of the node and the edge types are specified by the line type. Solid lines represent control dependence edges and dashed lines represent data dependence edges. Furthermore, the presence conditions (PC) and reaching conditions (RC) are depicted by a tuple `[PC, RC]`. If the tuple contains three dots (...), the element at this position does not matter.

Case 1. Introducing a presence condition on a control edge (Figure 5.5a) In this case a branch node tests a configuration condition. The incoming edge to node `IF f0` will only be executed if condition `C` is satisfied. However, the branch node further tests a configuration condition `f0`. Hence, the control dependence successors will only be executed if this node is executed and the generated presence condition is satisfied. Therefore, the outgoing edge leading to the node representing the then-branch is labeled with presence condition `C && f0`. Similarly, the edge to the node representing else-branch is labeled with `C` and the negation of `f0`.

Case 2. Joining presence conditions (Figure 5.5b) In this propagation case, a node



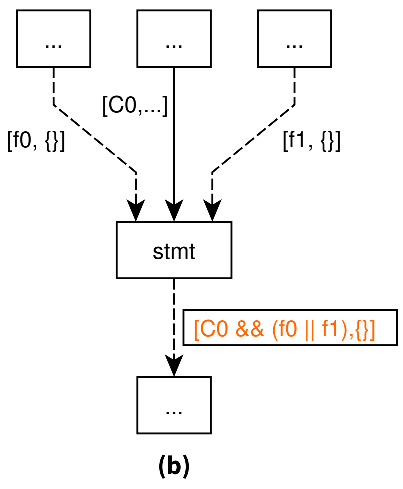


Figure 5.6: Propagation cases with interaction between presence and reaching conditions.
(a) Introducing and using new configuration variables. The reaching condition $(x, f0 \wedge f1)$ is used by branch statement `IF !x`. Therefore, a new presence condition is created by negating the reaching condition.
(b) Propagating presence conditions to data dependence edges. The outgoing edge will only be valid if the statement is executed, i.e., $C0$ is satisfied, and if at least one incoming data edge is valid, i.e., $f0 \vee f1$ is fulfilled.

has multiple incoming control dependence edges, each carrying distinct presence conditions. The conditions of the input edges are combined disjunctively and the combined condition is then applied to every outgoing edge since the node may be executed if either of the two control predecessors is executed.

Case 3. Introducing and using new configuration variables (Figure 5.6a) In this case, a reaching condition is transformed to a presence condition. A new configuration variable `x` is introduced, which stores a combination of two configuration conditions `f0` and `f1`. A data dependence edge propagates this definition to an if-statement using the variable `x` in its branch condition. Thus, the outgoing control edge is labeled with a presence condition `!(f0 && f1)`.

Case 4. Propagating presence conditions to data dependence edges (Figure 5.6b) This case shows how presence conditions on incoming control and data dependence edges are propagated to an outgoing data dependence edge. First, the presence conditions of incoming control dependence edges propagates to all outgoing data dependence edges, as the data dependence of the node only occurs if the node is executed. Therefore, the outgoing data dependence edge gets a presence condition C0 in the example. Secondly, there are incoming data dependence edges with presence conditions $f0$ and $f1$. However, the statement can only be executed if one of these edges provides data. The outgoing data dependence edge is thus only valid if the statement is

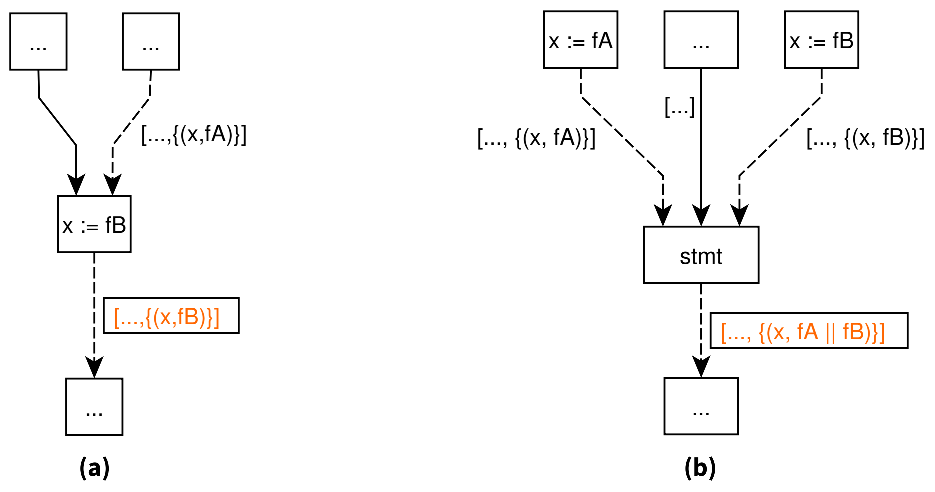


Figure 5.7: Basic cases of propagating reaching conditions.
(a) Updating a reaching condition set by a new configuration condition introduced by assigning configuration variable x.
(b) Joining reaching condition sets. The two incoming definitions of configuration variable x are combined disjunctively.

executed and if there is incoming data, i.e., the condition $C0 \wedge (f0 \vee f1)$ is satisfied.

Case 5. Updating reaching condition sets (Figure 5.7a) This case deals with propagating a reaching condition set across a node introducing a new configuration condition. The reaching condition set of the outgoing data dependence edge is updated with the new definition for variable `x`, i.e., `(x, fB)` overwrites definition `(x, fA)`.

Case 6. Joining reaching condition sets (Figure 5.7b) In this case a node has two incoming and one outgoing data dependence edge. The reaching conditions are combined using a union operation but reaching conditions for the same variables `(x, fA)` and `(x, fB)` are combined using a logical `OR` operator.

Algorithm 7 describes how the incoming presence condition and reaching conditions are computed. It implements the propagation cases. The input parameters are the CSDG, the current node, and the successor edge. The variable PCC is the presence condition function resulting from control dependence edge predecessors. The variable PCD is the presence condition function resulting from data dependence edge predecessors. The variable RCD is the reaching conditions function resulting from data dependence edge predecessors. In the first loop, the algorithm iterates over all incoming edges and depending on the predecessor edge's type, the presence condition and the reaching conditions are combined as illustrated by Case 2 and Case 6. For combining the reaching conditions function, the operator operator $\oplus$ is used. It forms the union of two given reaching condition functions $RC(e_0)$ and $RC(e_1)$ but definitions affecting the same variables are joined by combining the condition disjunctively. For example, $\{(x, fA)\} \oplus \{(x, fB)\} = \{(x, fA \vee fB)\}$ as illustrated in Figure 5.7b. After the for statement, the algorithm then tests if the node n contains an assignment statement and the receiver of the assignment is a variable contained in the reaching conditions. If so, the reaching conditions are updated such that (x, fB) is the only element using x on the left side in $RC(e)$ as (cf. Case 5). The else branch covers Case 3. It is tested if the node is an if statement and the expression contains a configuration variable. If so, the expression $expr$ is converted to a presence condition by replacing all occurrences of variables available in $RC(e)$. The resulting presence condition is then conjunctively combined with PCC. In the end of the algorithm, the new presence condition and reaching conditions are combined and returned. If the successor edge e is a control edge, the PCC is conjunctively combined with the seed condition (cf. Case 1). If the successor edge e is a data edge, the PCC is conjunctively combined with PCD (cf. Case 4) and the seed condition (cf. Case 1) and reaching conditions RCD are returned.

Algorithm 7 Computes the incoming condition for a provided edge.

```
procedure ComputeIncomingCondition($csdg = (V, E, T, C, PC, RC)$, $n$, $e$)
    $PCC \leftarrow \emptyset$                    ▷ presence condition for control dependence edges
    $PCD \leftarrow \emptyset$                    ▷ presence condition for data dependence edges
    $RCD \leftarrow \emptyset$
    for $pe = (p \xrightarrow{t} n) \in E$ do
        if $T(pe) = control$ then
            $PCC \leftarrow PCC \cup PC(pe)$
        else if $T(pe) = data$ then
            $PCD \leftarrow PCD \cup PC(pe)$
            $RCD \leftarrow RCD \oplus RC(pe)$
        end if
    end for
    if $n =$ "x := fB" $\wedge\ x \in Vars(RCD)$ then
        Update $RCD$ by inserting tuple $(x, fB)$.
        This replaces any tuple with $x$ on the left side.
    else if $n =$ "IF expr" $\wedge\ \exists var \in expr : var \in Vars(RCD)$ then
        Extract configuration condition from expression (cf. Section 3.3) and
        replace configuration variables by mapped values in $RCD$.
        Conjunctively combine this formula with $PCC$.
    end if
    if $T(e) = control$ then
        return $(PCC \cap getSeed(n), \emptyset)$
    end if
    if $T(e) = data$ then
        return $(PCC \cap PCD \cap getSeed(n), RCD)$
    end if
end procedure
```

5.3 Implementation

The tool *COACH!* implements the configuration-aware change impact analysis approach and was used to evaluate the approach on the industry partner's software product line KePlast. Recall, KePlast has been developed in a proprietary dialect of the IEC 61131-3 standard, for which no parser or compiler suitable for program analysis was available. Therefore, the analysis framework presented in Section 3.6, which specifically targets the programming language of the industry partner, builds the CSDG required for this approach.

The algorithm for performing CA-CIA has already been specified in the previous section (cf. Section 5.2) using pseudo code. Hence, this section will focus on the description of some vital performance optimizations.

Explicit SDG representation Building a SDG on the level of Jimple statements blows up the number of required statement nodes in the SDG. To cope with this issue, the CSDG representation was implemented to implicitly represent statement nodes in memory. The implicit SDG implementation leverages basic blocks from the CFG to reduce the number of control dependencies to store. Furthermore, intraprocedural edges and statement nodes are created on demand to safe memory. The drawback of this implementation is that traversing the SDG is more expensive in terms of run-time performance. To boost run-time performance, it is necessary to implement a CSDG explicitly representing all edges and statement nodes.

Shortcut evaluation As it is usually done in most programming languages, COACH! uses a shortcut evaluation when computing the incoming condition. In particular, if the control predecessors are combined disjunctively and one of the predecessors carries condition *true*, the iteration can be aborted. This optimization helps a lot especially for heavily used library nodes that are often in the processing fringe with thousands of control predecessors.

Method collapsing If a method is known to not alter the variability information, i.e., not containing any nodes with seed conditions and not using reaching conditions, COACH! does not propagate the variability information through the method but moves the condition directly to the outgoing edges to save many intermediate propagation steps.

5.4 Evaluation

The evaluation investigates the benefits regarding complexity reduction and the performance of the CA-CIA approach in two use cases: (i) development and maintenance in domain engineering, i.e., for determining the different product variants affected by a change; and (ii) development and maintenance in application engineering, i.e., for determining code affected by a change made to a specific product variant. Specifically, we use product families of the industry partner to explore the following research questions:

RQ1. *How beneficial is the configuration-aware CIA for maintaining a product line?* The hypothesis investigated by this research question is that the approach is more beneficial the higher the variability complexity is. In the worst case it could happen that only trivial variability information, i.e., presence condition *true*, is available. In this case, the approach is not helpful at all and RQ1 evaluates this question. We thus estimate the benefit by computing the degree and complexity of variability information a domain engineer needs to consider when determining the impact on product variants after changing code in a product line.

RQ2. *How beneficial is the configuration-aware CIA for maintaining a specific product variant?* This situation is common in clone-and-own product lines. Some of the variability has already been resolved, e.g., by setting configuration options. However, developers still need to adapt and fine-tune the product variant to meet specific customer needs. We estimate the benefit by computing the increased precision of the change impact, as a smaller change impact will reduce the development effort.

RQ3. *Is the performance of the configuration-aware CIA sufficient for realistic maintenance tasks?* A major problem of static program analysis is high run-time complexity. We perform benchmarking to show the practicality and suitability of the approach in realistic maintenance tasks.

5.4.1 Case Study and Code Base Selected for the Evaluation

To investigate these research questions, we applied our tool COACH! to the industry partner's product line KePlast. Recall the development process applied by KEBA's developers for KePlast (cf. Section 1.2). KEBA uses a custom-developed product configuration tool to select components from the KePlast platform based on customer requirements. The selected components are then adapted and extended by application engineers to meet specific customer needs not yet covered by the platform. In this stage, the derived software is still configurable by using load-time configuration options, the variability mechanism targeted by the CA-CIA approach.

As a baseline for the evaluation, product variants from KePlast's families which contain a maximum number of features selectable together in the product configuration tool are derived. These maximum products do not contain the full code base of the product line, but are an approximation good enough for the purpose of this study. In particular, we used 9 different still configurable product variants with a size ranging from 53 kLOC (*family4*) to 302 kLOC (*family2*).

Furthermore, these maximum product variants of a family indeed have commonalities and differences. In the case of the industry partner's product families, the common code implementing the mandatory features is a substantial part of the source code because it includes implementations for basic data structures and algorithms used in all product variants. The evaluation therefore concentrates on the variable part of the analyzed product families to obtain data that is most relevant for the research questions. Specifically, change impacts containing at least one presence condition are considered. This was the case for 27% of all change impacts in our case study (median across product families).

5.4.2 RQ1 – Domain engineering

As argued before, the benefit of the configuration-aware CIA depends on the complexity of the variability information and the contradictions of the presence conditions, which allow pruning the change impact. Recall that in the CA-CIA approach variability information is propagated based on the program dependencies and is therefore usually not visible directly in the source code.

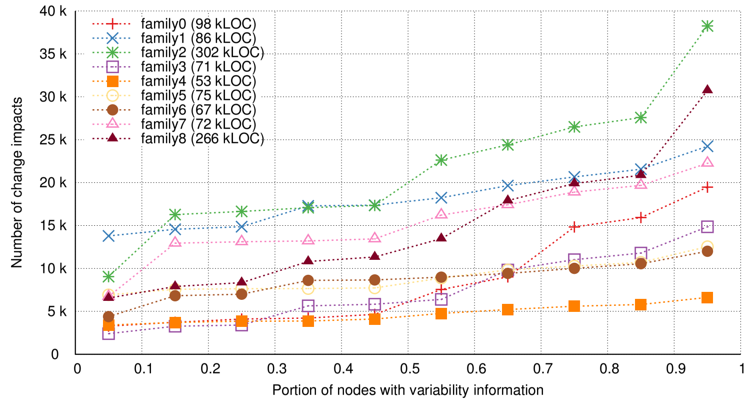
We define three *metrics* to characterize the variability of a code base.

The *Variability Complexity* is the ratio of edges with variability information to the total number of edges within a change impact. CA-CIA is more beneficial if more edges are annotated with variability information, as this would make a manual CIA even harder.

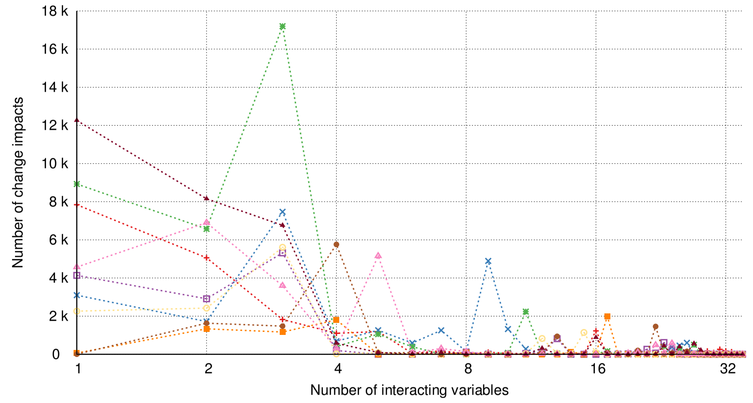
The *Variability Interaction Order* measures the average number of involved distinct configuration options in the variability information. This is computed by counting the number of distinct configuration options in a single presence condition and computing the mean of these numbers over all edges. For example, if the two presence conditions $a \wedge b$ and $\neg a \vee c$ are in a change impact, the interaction measure would be $\frac{2+2}{2} = 2$. This metric thus represents the interaction between different configuration options. The benefit of the approach increases with the order of the variability interaction, as manually analyzing complex interactions is hard to infeasible.

Contradicting Conditions Ratio. The propagation of presence conditions may result in contradicting presence conditions, which can never become valid, regardless of the product configuration. Such invalid edges allow removing statements from the change impact. The measure is the ratio of invalid edges to the total number of edges. A higher value is better because the change impact is smaller and more precise.

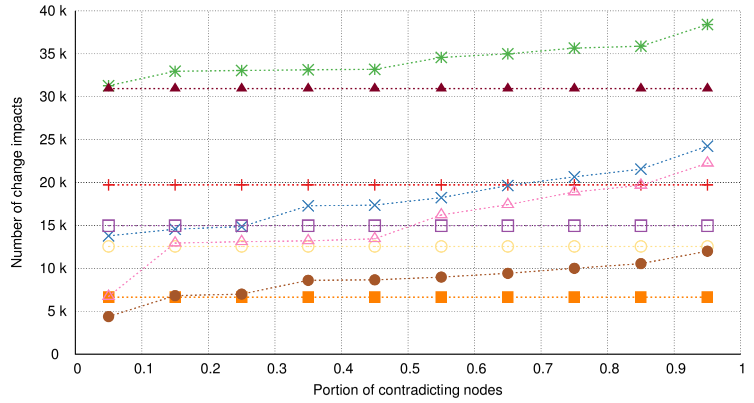
Method. COACH! performed CA-CIA for every single statement in the selected product families. This was done in two phases: in phase 1 we built the CSDG for the product family. This included parsing the source code, performing control flow, data flow and pointer analysis to build the SDG, and extracting the initial conditions from the source code. In phase 2 it performed the configuration-aware change impact analysis as described in Section 5.2.1. Specifically, COACH! iterated over *all* statement nodes in the CSDG and performed a configuration-aware CIA.



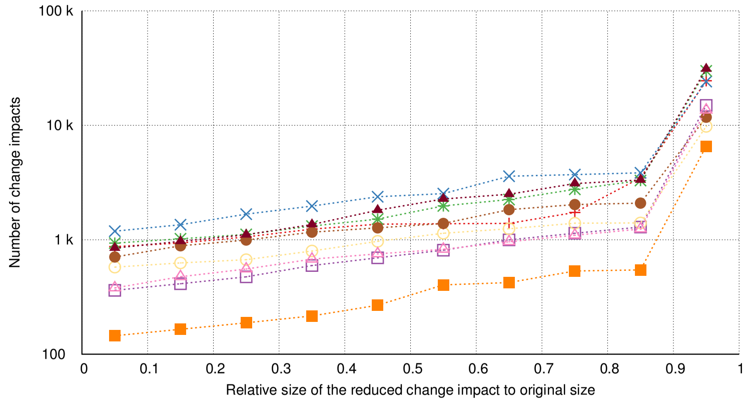
(a) RQ1 – Variability Complexity.



(b) RQ1 – Variability Interaction Order.



(c) RQ1 – Contradicting Conditions Ratio.



(d) RQ2 – Relative Change Impact Size.

Figure 5.8: Results for RQ1 and RQ2 computed by applying our tool to a set of real-world product families provided by the industry partner.

Results. Figure 5.8a shows the results of the evaluation for the metric *Variability Complexity*. The x-axis shows the complexity values, i.e, the portion of nodes with variability information. The values are grouped into intervals of width 0.1. Thus the x-axis represents intervals $]0.0, 0.1[$, $[0.1, 0.2[$, …, $[0.9, 1.0]$ and an interval is denoted by its middle value. The y-axis is the number of change impacts contained in a certain interval. The chart shows the cumulative distribution of change impacts for these groups. For example, the data point at 0.65 of the *family2* says that approximately $25,000$ change impacts had a variability complexity value in interval $]0.0, 0.7[$. Therefore, a presence condition was available for up to 70% of the statements in the change impact.

Figure 5.8b shows the results for metric *Variability Interaction Order*. The x-axis shows the average number of configuration options involved in the presence conditions of a change impact. The y-axis is the number of change impacts that have a certain average number of involved configuration options. For example, the *family6* has a peak at $x = 4$, i.e., around $6,000$ change impacts involve on average 4 configuration options.

Figure 5.8c shows the results for metric *Contradicting Conditions Ratio*. The x-axis shows the portion of nodes that are invalid because of a contradicting condition. The values are again grouped into intervals of width 0.1. The x-axis therefore represents intervals $]0.0, 0.1[$, $[0.1, 0.2[$, …, $[0.9, 1.0]$ and again, an interval is denoted by its middle value. The y-axis again represents the number of change impacts as in Figure 5.8a. The chart shows the cumulative distribution of change impacts for these groups. For example, in *family2* at $x = 0.15$ the size of $33,000$ change impacts could be reduced by up to 20%.

Discussion. The results of metric *Variability Complexity* shown in Figure 5.8a show that the approach is beneficial given the complexity of variability in real-world systems. The graph shows a cumulative distribution function. The steeper a line the more variability information is available in the computed change impacts. The chart also shows that larger product families provide even more variability information. For example, consider *family2*, the largest product family in our study. It first grows moderately until 0.45 but then it starts to grow faster, i.e., 50% of the statements in most change impacts had variability information available. When combining the data in Figure 5.8a to one value, we see that overall the change impacts have 50%-60% presence conditions available on average (median). We therefore conclude that CA-CIA is even more beneficial in larger systems, as more variability information needs to be considered.

The results for metric *Variability Interaction Order* shown in Figure 5.8b show a similar picture. Dealing with interactions involving 2 or 3 configuration options is already quite cumbersome. The results show that the majority of all change impacts had presence conditions with up to 5 configuration options involved on average. Other empirical studies have shown an interaction degree of 2 or 3 to be common [Apel13], similar to these results. However, there were also quite a few change impacts with higher numbers of 10 to 35 configuration options per presence condition, an order that is almost infeasible to comprehend by developers.

Section 5.4.1 describes that the analysis is performed on product families which have already been partially configured and can be compiled. Therefore, the product families' source code does not contain any mutually exclusive feature implementations. So regarding the *Contradicting Conditions Ratio* we did not expect a significant increase of the precision of change impacts by finding contradicting presence conditions. This did in fact happen for the analyzed product families *family0*, *family4*, and *family5*. The corresponding lines in Figure 5.8c do not increase, i.e., no statements could be removed from the change impacts. However, the precision of change impacts in other product families could still be increased. For example, the difference between the last two data points of *family2* shows that it was even possible to find approximately $2,000$ change impacts whose size could be reduced by 90–100%.

5.4.3 RQ2 – Application engineering

For this use case we perform CIA for a specific product configuration, i.e., we remove elements from the result that have not been used in this specific configuration. Again, we distinguish between change impacts with and without variability information, because the latter one covers most commonly just library code, which is less interesting in terms of variability. We define metric *Relative Change Impact Size* to measure the reduced numbers of edges in the change impact after evaluating the presence conditions compared to the original change impact. The original change impact is the configuration-aware change impact but ignoring the presence conditions.

Method. Analogously to the method for RQ1, we performed the configuration-aware CIA for every single statement of our product families (phase 1). However, for answering RQ2, we created concrete product configurations by randomly generating Boolean values for each known configuration option (phase 2). When computing the change impact, the presence conditions were then evaluated using these randomly generated values for the configuration options. We measured the reduction of the change impact size compared to its original size. This was repeated 10 times with different generated values to compute an average for the *Relative Change Impact Size*.

Results. Figure 5.8d shows the results for RQ2. Each line corresponds to one of the analyzed product families. The x-axis and y-axis represent the same as in Figure 5.8a. The results are again grouped into intervals of width 0.1. For example, the data point at $x = 0.85$ of *family4* means that the size of approximately 800 change impacts has been reduced by 10% to 20%.

Discussion. We observe that evaluating the presence conditions after computing the change impacts reduces the size to 90% in most cases. Figure 5.8d shows a sharp edge at the end of all lines, which means most change impacts are in the last group. We expected these numbers to be more distributed across the other groups. But there are also many change impacts that could be reduced to less than 80% of their original size. To answer RQ2, we conclude that evaluating the presence conditions yields noticeable benefit for maintenance because fewer statements have to be considered, although we expected to do better.

5.4.4 RQ3 – Performance

One major problem of static program analysis techniques is commonly their analysis time. We therefore measured the analysis time of our tool when analyzing the industry partner's product families.

Method. The performance evaluation needs to consider both building the CSDG and performing the configuration-aware CIA. We already showed in Section 3.7 that building the CSDG, including the time needed to do all required analyses, is in a nearly linear relation to the size of the analyzed software. The same is true for the peak memory consumption. In this evaluation, we thus report performance results for the technical contributions of the CA-CIA. We therefore measured the size of all configuration-aware change impacts and the time required to compute them (cf. phase 2 of RQ1 and RQ2).

Results. The results of this evaluation part are shown in Figure 5.9. A cross in the chart represents a change impact. The x-axis lists the size of the change impacts in terms of included statements. The y-axis shows the time required to compute the change impacts in milliseconds. The performance evaluation has been executed on an Intel Core i7-4770, 3.40 GHz, 16 GB DDR3-RAM machine running MS Windows 7 64-bit.

Discussion. The results show that the computation of a change impact is fast and never takes longer than 3.5 seconds. It also seems that the time required to compute a change impact does not depend on its size because we cannot observe any linear or higher-order dependency on the change impact size. We assume the time is dominated by the first step of the CA-CIA algorithm, i.e., computing the backward slice to determine the possible influencing presence conditions. However, computing the change impact is negligible compared to the time required to build the SDG which took between 30 and 270 seconds (cf. Section 3.7).

5.4.5 Threats to Validity

There is a potential bias caused by the selection of product families in a specific application domain that have been developed in a specific programming language. This section thus present detailed evaluation results and avoid generalizations of how well the approach would work for other programming languages and PLs. However, the

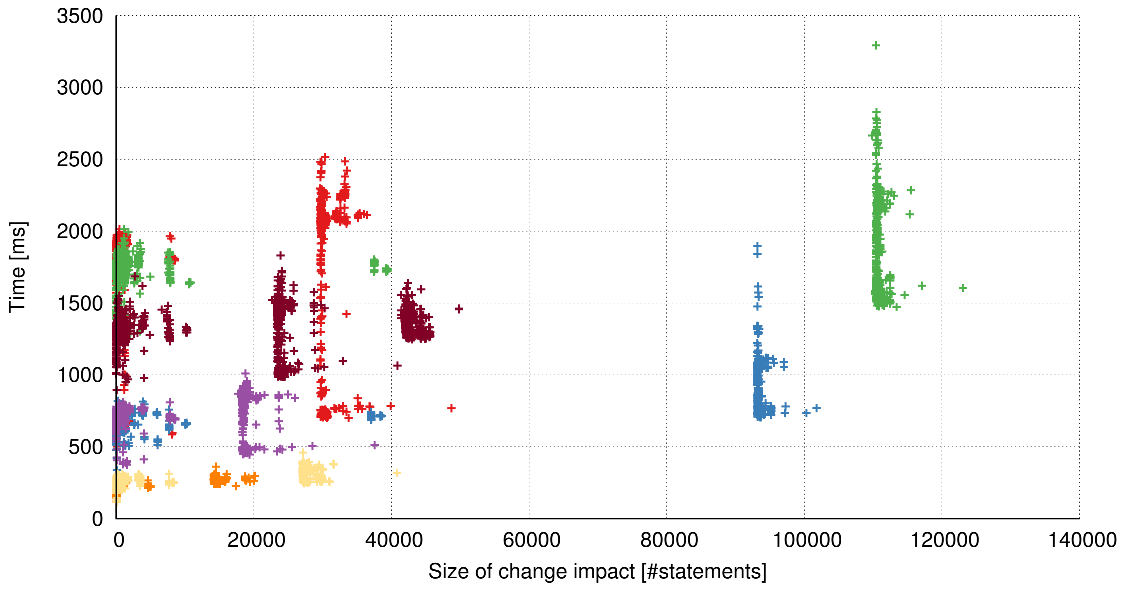


Figure 5.9: RQ3 – Average time required to compute configuration-aware change impacts of a specific size.

evaluation focuses on load-time configuration options, a variability mechanism that is widely used in all programming languages. Also, given that companies typically do not provide access to data about their product lines we believe that our evaluation results are valuable and promising.

Specifically, the evaluation is based on partially configured product families, that have been derived from a product line by selecting all features defined in a custom-developed configurator. COACH! could not analyze the full code base of the product line, as the variability information is stored implicitly in the configurator. As a result the evaluation is based on a less configurable code base, e.g., certain alternative features are not included. However, this means that the results would be even more favorable for the full code base.

A prerequisite of the approach is that the mechanism for implementing variability in source code is known. This could be a problem in software projects without any conventions for this aspect, as extracting the initial conditions might not be possible. However, related work [Berger10] indicates that most of the time it is known how variability is realized. Furthermore, Reisner et al. [Reisner10] show that open source systems heavily use configuration options to implement variability.

The analyzed product families are executed in a runtime environment that provides library functions, whose implementation is unknown. This is also a source of imprecision when building the CSDG. However, such system functions occur in almost all execution environments and must be handled appropriately. Our implementation handles this problem by assuming the worst case if something is not known. For example, if a system variable returns a reference to a variable and this reference cannot be determined, we assumed that every reference may be returned. This is the common strategy to preserve soundness but sacrifices precision.

Several measures have been taken to mitigate the risk of incorrect computations: the data flow and pointer analysis of the widely used Soot tool suite are used to build the SDG, so there is a high confidence that this part of the implementation produces correct results. Furthermore, the part of the implementation that extracts variability information from source code has already been reviewed by a developer of the industry partner in a qualitative process as reported in the evaluation of Chapter 4.

5.5 Discussion

This chapter presented the CA-CIA approach which is the main application of the delayed variability analysis concept in this thesis. The problem illustration (cf. Section 5.1) the need for an analysis that is able to handle load-time configuration. But there are additional advantages and disadvantages of our approach we want to discuss in this section.

The delayed variability analysis concept allows to implement analyses based on existing implementations. This is an intrinsic property because instead of modifying the analysis itself, variability information is added afterwards. Therefore, the effort for implementing a variability-aware analysis can be quite low. Of course, some preconditions must be fulfilled and the most important one is that results of the analysis must be a graph data structure that represents control dependencies, i.e., the control dependence subgraph (CDS). However, since the CDS can be built using standard data-flow analysis, this restriction is not problematic because this is available for most programming languages.

A further advantage of the delayed variability concept is that it enables specific treatment of dependency types. In the case of CA-CIA, data dependencies are handled in a special way to leverage the information how configuration values are propagated. In this way, it is possible to improve the analysis by doing an additional but very lightweight analysis on top of the result. CA-CIA uses the information of propagated configuration values to improve the precision by introducing new seed conditions (cf. Figure 5.6a in Section 5.2.2). Other special treatments for other types of dependencies are of course possible but have to be specified for the particular analysis.

5.6 Related Work

The concept of delayed variability and the CA-CIA approach are related to work concerning (i) variability-aware program analysis and (ii) run-time variability and load-time configuration options.

Variability-aware program analysis. Several approaches use information about variability for eliminating redundant computations in variability-aware program analysis. For instance, the approach by Kästner et al. [Kästner11] parses preprocessor-based product lines written in C, i.e., unpreprocessed C source code, to build a variability-aware abstract syntax tree (VAST) using presence conditions to represent variability. The approach also provides a variability-aware type checker on top of the VAST. Liebig et al. [Liebig13] implemented two variability-aware analyses, i.e., type checking and variable liveness analysis, for analyzing the Linux kernel, a very large preprocessor-based product line. The authors also implemented the variability-aware monotone framework to demonstrate that variability-aware approaches are able to handle large-scale product lines. Brabrand et al. [Brabrand12] describe how to make an intraprocedural data-flow analysis variability-aware by adding propositional formulas to intermediate results during analysis. All these approaches extract variability information in the beginning of the analysis chain and consider this information in every subsequent analysis step. For example, it is common to build a variable AST and a variable control flow graph to then carry out the variability-aware analysis. Considering variability throughout these intermediate steps allows to implement analyses that are as precise as the *brute force* approach of analyzing all product variants individually, which is usually taken as the ground truth to compare to.

The work presented by Midtgaard et al. [Midtgaard14] further generalizes the lifting framework of Brabrand et al. [Brabrand12] to be applicable to all analyses that can be represented with abstract interpretation. The presented approach is currently not automated but describes how to make an analysis variability-aware in a systematic way.

Run-time variability and load-time configuration options. The idea of delayed variability-aware program analysis was proposed originally to support program analysis in product lines using load-time variability mechanisms. For instance, Lillack et al. [Lillack14] present an approach for tracking load-time configuration options by using a taint analysis to determine source code depending on configuration options and to compute presence conditions for the code. Xu et al. [Xu13] introduce a tool named SPEX that analyzes a program's source code to infer constraints for configuration options by tracking the data-flow of each configuration variable. They use program slicing to reduce the domain to consider but only on a configuration variable's data-flow graph. The reduced domain is then used to infer constraints for configuration options by analyzing the statements using them. This is related to the CA-CIA approach, as we also analyze how configuration variables are used. However, SPEX concentrates on reverse-engineering of a variability model instead of computing the influence of variability on the program. Reisner et al. [Reisner10] investigates the effect of configuration options to the behavior of programs in an empirical study. They use symbolic evaluation to find out how settings of run-time configuration options affect line, basic block, edge, and condition coverage of test suites. We also compute the influence of configuration options to statements. However, the CA-CIA approach is not limited to compute the impact when changing a configuration option's value but works with arbitrary statements.

5.7 Summary

This chapter introduced the configuration-aware change impact analysis (CA-CIA) approach as one of the core contributions of this thesis. The problem illustration showed that configuration influences the control and data flow in programs, thus motivating CA-CIA.

CA-CIA uses the delayed variability concept (cf. Section 1.1) to make program slicing based CIA configuration-aware using the CSDG. It propagates presence conditions representing the variability of the analyzed software system.

The main part of this chapter described the CA-CIA by specifying the algorithms in pseudo code. CA-CIA implements an incremental propagation approach for avoiding global propagation within the of variability information within the whole CSDG. Furthermore, six basic propagation cases illustrated how presence conditions are combined and propagated.

The description of the implementation discussed some crucial optimizations, i.e., explicit representation of the CSDG, shortcut evaluation, and method collapsing, which were required to make CA-CIA fast and applicable for large-scale systems.

The CA-CIA approach was evaluated regarding the benefit and performance using 9 product families of the industry partner KEBA. It investigated in detail the complexity and distribution of variability information and the required analysis time. The evaluation of the CA-CIA approach shows, that variability has substantial influence on the programs control flow. Specifically, it was observed that in more than 50% of all computed change impacts, half of the impacted nodes carried variability information. The results also show that the interaction degree of configuration options is high, further justifying the importance of a configuration-aware approach. Specifically, interaction orders involving up to 35 configuration options were observed. The run-time performance showed that the technique scales for industrial systems.

Chapter 6

Compositional Change Impact Analysis for Configurable Software

Software systems are usually designed and implemented in a modular way to address challenges such as complexity, multi-language development, distributed development, and continuous and long-term evolution. When analyzing modular systems, one would like to exploit the modular structure of the system to perform the analysis in a compositional way. However, if the analysis uses global information, performing an analysis just on modules independently from other modules is difficult. For example, variability is often a global information and needs to be addressed accordingly.

This chapter is based on [Angerer16] and presents the CAM-CIA approach that builds on the CA-CIA technique presented in Chapter 5. Specifically, the challenge of modularizing the analysis lies in correctly handling the variability information represented by Boolean conditions.

Since the CSDG is built by adding interprocedural dependencies between PDGs, modularizing the CSDG could be done by cutting these dependencies. However, the CA-CIA technique cannot simply be modularized in this way as it propagates variability information across the CSDG when computing a change impact. Cutting interprocedural dependencies will lead to incorrect results and therefore modularizing the analysis requires treatment of variability information. The CAM-CIA approach addresses this problem and is intended to support program analysis in large-scale software product lines. It is common in product lines to create product variants by composing modules in a staged configuration process as explained in Section 1.2. In such a development context program analysis can exploit the modularity of the system by first analyzing individual modules, and later reusing and composing the pre-computed analysis results. The approach enables incremental CA-CIA, therefore reducing the analysis effort if changes just affect a single module.

The contributions of this chapter are

(i) an approach for compositional CA-CIA using placeholders to modularize variability information,

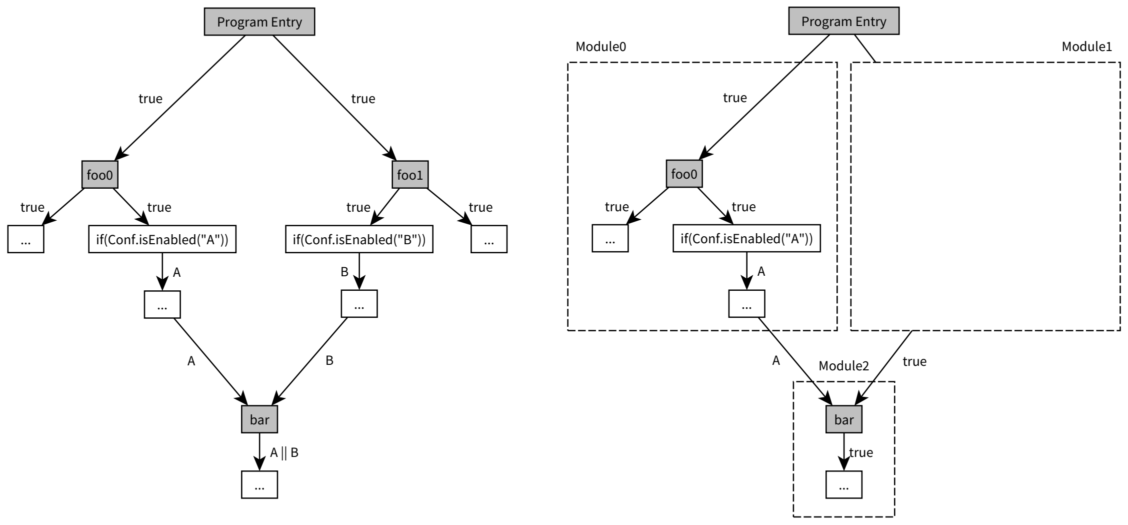


Figure 6.1: Partial CSDG demonstrating the problem of cutting dependencies carrying variability information during analysis. The dashed lines indicate module boundaries.

(ii) support for composing pre-computed analysis modules, and

(iii) an experiment showing that the compositional CIA produces the same results as the non-compositional version and can be applied in an incremental manner.

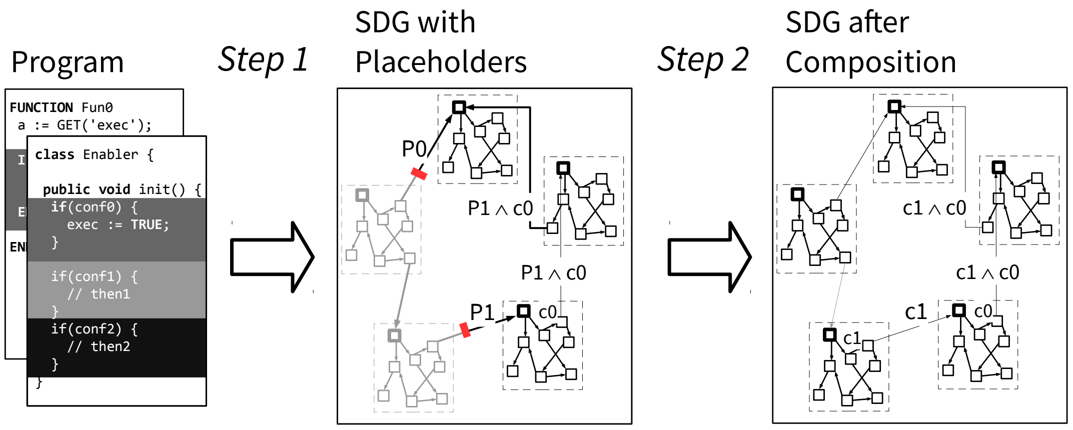
The remainder of this chapter is organized as follows: Section 6.1 illustrates the problem of modularizing the CSDG and in particular the variability information. Section 6.2 explains the modularizing approach based on placeholders for conditions. Section 6.3 explains the setup of our evaluation and discusses its results. Section 6.4 discusses related work and Section 6.5 concludes this chapter.

6.1 Problem Illustration

Performing CA-CIA in a compositional way is challenging as it requires a backward slice (cf. Chapter 5) to determine all possibly influencing nodes as simply stopping backward slice at a module boundary would reveal invalid results. Invalid in this context means that a result differs from the corresponding result in the monolithic version of the analysis. Thus, a way to handle the information missing from other modules is needed. Compositional CA-CIA can only be achieved by modularizing the underlying data structure, i.e., the CSDG. Building the CSDG involves two phases:

1. Performing control and data flow analysis for each method resulting in PDGs.
2. Linking the PDGs by inserting interprocedural dependencies representing calls or data flow over parameters, global variables, and so on.

Therefore, a straightforward strategy to modularize the CSDG is to cut the interprocedural dependencies gaining *analysis modules* at the granularity of methods. Figure 6.1



```
FUNCTION Fun0
 a := GET('exec');
```

```
class Enabler {

 public void init() {
  if(conf0) {
    exec := TRUE;
  }

  if(conf1) {
    // then1
  }
  if(conf2) {
    // then2
  }
}
```

Figure 6.2: Overview of the CAM-CIA approach: in Step 1, the configuration-aware CIA is performed on the available modules. Placeholders (P0, P1) are introduced for unavailable modules (shown in gray). In Step 2, the partial results are composed by loading missing modules and replacing concrete values for the placeholders.

shows a partial CSDG with variability information. The dashed lines indicate module boundaries and a module is in this case a method. Assume we want to do an analysis in `Module2` and therefore need to know the variability information reaching this module. Method `bar` is called by methods `foo0` and `foo1` introducing conditions A and B, respectively. If `Module1` and the corresponding variability information are missing, we need to assume that the default condition *true* flows in. Therefore, instead of resulting in condition $A \vee B$ we just would gain *true*. However, this is different to what the analysis would reveal if there are no modules and the result is therefore invalid. Nevertheless, condition *true* is not unsound but a crude over-approximation and therefore not useful at all.

6.2 Approach

Figure 6.2 depicts a high-level view of the two-step CAM-CIA approach:

Step 1 CAM-CIA first performs a modified configuration-aware change impact analysis that introduces placeholders to support modularity. It stops traversing the SDG during a forward or backward slice at defined barriers – the boundaries of the program modules that are analyzed independently. This could, for example, be a method, a package, or a subsystem. Therefore, the approach introduces a condition placeholder if it reaches a barrier. The condition placeholders serve as substitutes for the variability information not known in this phase, i.e., if slicing stopped due to a barrier. The placeholder is then used during forward slicing instead of the concrete condition. It is crucial to handle the missing variability information especially during backward slicing, which determines the presence condition for a statement (cf. Section 5.2.1).

The placeholder-based approach is shown in Step 1 of Figure 6.2. The SDG consists of independent PDGs with interprocedural dependencies. The PDGs are shown as boxes with dashed borders. However, the PDGs may not be available (shown in gray), for example, if currently not loaded. Therefore, the interprocedural link would be considered as interrupted. During the backward slicing phase of the CA-CIA, the placeholder variables `P1` and `P2` are introduced if interrupted links are encountered. These placeholders are further used during forward slicing as a replacement for the still missing information.

Step 2 In the composition phase, the approach computes the concrete conditions for placeholders and replaces them accordingly. This is done by loading the PDG of the module causing the placeholder to be introduced. Two cases need to be distinguished once the PDG has been loaded: (1) if the PDG has already been analyzed, the concrete presence condition for the placeholder is already known and can simply be replaced; (2) if the PDG first needs to be analyzed, the concrete presence conditions are computed by performing a backward slice at the corresponding nodes. The nodes can only be method nodes or formal parameter nodes because such nodes form the *interface* of a PDG. For example, in Figure 6.2, Step 2 the placeholder `P1` is replaced by the concrete condition `c1`. This replacement is performed for all presence conditions that use the placeholder `P1`. In this way, it is not necessary to recompute the variability information since the result is valid with respect to the available modules.

6.3 Evaluation

The evaluation investigates the correctness of the CAM-CIA approach and its benefits compared to the non-compositional configuration-aware change impact analysis. Specifically, the following two research questions are explored using product families of our industry partner.

RQ1. *Are the results of the CAM-CIA correct compared to the non-compositional version?* The results of the compositional analysis are compared to the results of the non-compositional CA-CIA. Specifically, the non-compositional CA-CIA (cf. Chapter 5) produces the ground truth and compare its results to the one of the CAM-CIA.

RQ2. *What is the benefit of the compositional CAM-CIA compared to CA-CIA?* The CAM-CIA approach benefits during composition from situations where the condition placeholders can be eliminated, e.g., due to a contradiction or tautology. In this case, the variability information of other analysis modules would not be required. To answer this question, we count the number of cases in which this situation occurs.

6.3.1 Experiment Subject

We applied the approach to KEBA's KePlast software system. One product variant containing the maximum number of features selectable together in the product configuration tool was derived from the KePlast platform as a baseline for the evaluation. This maximum product does not contain the full code base of the product line but is an approximation good enough for the purpose of our study. Within KePlast, the component *Mold* serves as a starting point for the CIA. Its size is about 13,000 LOC in KEBA's domain-specific programming languages, translating to 43,500 Jimple statements.

6.3.2 Evaluation Strategy

To answer RQ1 and RQ2, we used the same evaluation strategy as for CA-CIA approach (cf. Chapter 5), i.e., computing the configuration-aware change impact for every single statement in the evaluation subject. The evaluation driver performed the CAM-CIA and CA-CIA for every single Jimple statement in the selected component. This was done in two phases: Phase 1 built the CSDG. This included parsing the source code, transforming the code to Jimple, building the SDG, and extracting the initial conditions from the source code. Phase 2 performed the compositional CA-CIA and the non-compositional CA-CIA. Specifically, we iterated over all method nodes of component *Mold* and all statement nodes for each method node. Every statement node was then taken as a CIA criterion for the compositional and non-compositional CA-CIA. While we computed the change impacts starting with statements in component *Mold*, many resulting change impacts obviously contain statements from other components of the KePlast product line. Our initial tests when preparing the experiment showed that about one third of the change impacts contain the entire system. This is possibly the result of reaching central nodes in the KePlast architecture. We decided to exclude these change impacts in our evaluation, as they are not useful for developers.

To answer the research questions RQ1 and RQ2, we define following metrics: CI is the total number of computed *change impacts*, i.e., the number of statements taken as CIA criterion. We compute the change impacts using the compositional and non-compositional CA-CIA for every CIA criterion and compare the two results. ECI is the number of *equal change impacts* for the compositional and non-compositional CA-CIA (cf. RQ1). If ECI equals CI we can assume the compositional CIA to be correct. PNR (*placeholders not required*) is the number of change impacts where placeholders disappeared during change impact analysis and did not have to be resolved during composition. Regarding RQ2 the metric PNR allows estimating the number of cases for which we can omit Phase 1 of the composition, i.e., computing the backward slice and performing a condition propagation to determine the concrete formula for the placeholder. In other words, PNR represents the number of change impacts that are more easy to resolve in the composition phase as a result of our compositional approach.

6.3.3 Results

We computed the compositional and non-compositional configuration-aware change impact for CI=23,124 Jimple statements of *Mold*, excluding the change impacts that contained the entire system as described above. The compositional and non-compositional variants matched in all 23,124 cases, confirmed by the ECI value of 23,124. This suggests that our compositional approach works correctly (RQ1).

Regarding RQ2 we grouped the change impacts by their size (the number of nodes) as shown in Figure 6.3. The chart shows the total number of change impacts for the different groups (height of the bars), and the $PNRs$ for the different groups (black bars), i.e., the change impacts requiring no resolution of placeholders. We omitted the 15,870 change impacts with less than 3 nodes in the chart.

The fractions of $PNRs$ are relatively high for all groups. This is not surprising for small sizes of CIs as they will mostly contain local nodes. However, the results also show high fractions of $PNRs$ for large change impacts, demonstrating the benefit of our approach.

The main threats to validity are related to run-time performance. Our experiment demonstrated scalability of the approach for the change impact sizes reported in Figure 6.3. However, we encountered problems in the composition phase of our approach for change impacts including the entire system. Upon investigation, we discovered that this is caused by the used implementation of presence conditions, which is based on *binary decision diagrams (BDDs)*. While BDDs, however, are very fast for logical operations, replacing a variable in a BDD is a costly operation. Thus, for overcoming this problem we are considering alternative implementations for presence conditions support more efficient replacements of variables. However, this issue did not affect the results of our experiment in any way.

6.4 Related Work

The problem of modularization has already attracted attention in related areas such as object-oriented program slicing and cross-language analysis. Larsen et al. [Larsen96] developed an program slicing approach for object-oriented programs, which uses the object structure to modularize the analysis. Similar to our work, their approach enables the reuse of previously computed analysis results. Korel et al. [Korel98] perform program slicing at the level of modules for better understanding the slices. Their approach reduces the size of slices by using modules as abstraction. Cross-language program analysis approaches also need to modularize the analysis as they

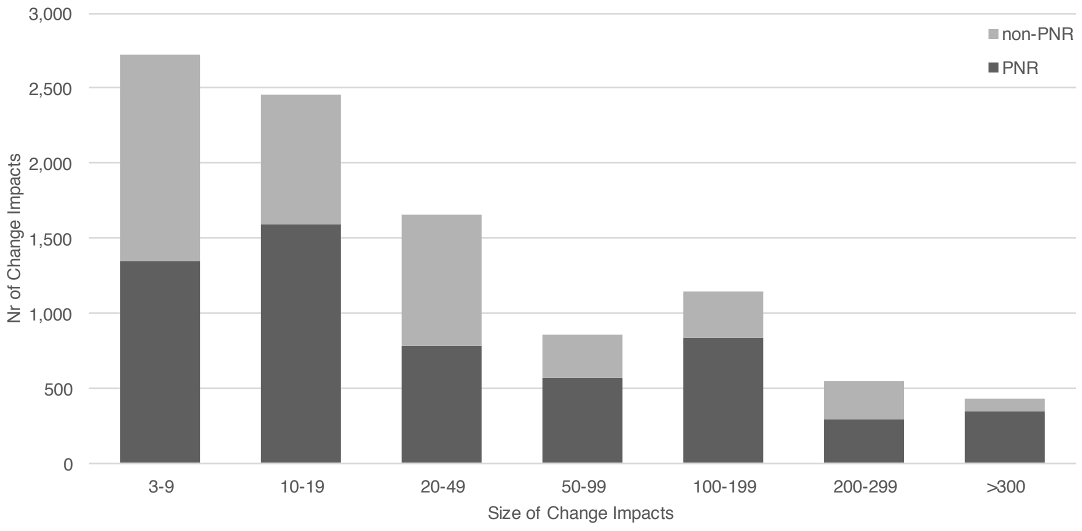


Figure 6.3: Results for metric PNR for different sizes of changes impacts.

have to deal with multiple subsystems written in different programming languages. Strein et al. [Strein06] address this problem in the context of software refactoring. Their analysis approach integrates all language modules into one common model representing the whole system. The *XLL* approach by Mayer et al. [Mayer12] relies on explicit communication links between independent models. Each model represents one language part and the parts are connected by explicitly defined links. Program analysis then exchanges information if such links are visited.

6.5 Summary

This chapter presented the CAM-CIA approach for performing compositional CA-CIA using program dependence graphs as modules. The work addresses the challenge of correctly handling global variability information that is required before the actual change impact analysis can be computed. Placeholders are used to deal with missing variability information at module boundaries. It has been shown that the CA-CIA can be performed within a module without restrictions, resulting in sets of nodes and edges where the edge conditions may contain placeholders. CAM-CIA also described how to compose analysis modules by resolving placeholder variables.

The evaluation investigated the correctness of the CAM-CIA approach and its benefits compared to the non-compositional CA-CIA approach. It therefore performed both analysis for 23,124 statements in the KePlast product family. The results show that the CAM-CIA gives the same results after the composition of modules.

Chapter 7

Comparing Lifted and Delayed Variability-Aware Program Analysis

Section 2 already discussed the difference between lifted analysis and delayed variability analysis approaches. This chapter presents results of an in-depth comparison of the two tools SPL$^{\text{LIFT}}$ and COACH! implementing the different analysis strategies.

SPL$^{\text{LIFT}}$ [Bodden13] is an implementation of the lifted strategy that is based on Brabrand's lifting framework [Brabrand12], while COACH! (cf. Chapter 5) uses the delayed strategy. While both tools have been shown to be useful, a systematic comparison to investigate their benefits and trade-offs is still missing. This chapter describes an in-depth experiment comparing the performance and precision of the tools SPL$^{\text{LIFT}}$ and COACH!. It report results and lessons learned intended for developers and software analysts that need to develop or select a variability-aware program analysis technique for a certain purpose or development context.

Although the two tools share the same goal – adding variability information to analysis results – they still cannot be compared directly. It was thus necessary to develop an experiment design enabling the comparison of reaching definitions analysis. Specifically, the contributions described in this chapter are: (i) an automated approach that uses SPL$^{\text{LIFT}}$ to build a variability-aware system dependence graph; and (ii) results, experiences, and lessons – in particular regarding performance and precision – gained from applying SPL$^{\text{LIFT}}$ and COACH! during the experiment to a number of SPLs.

This chapter is structured as follows: Section 7.1 introduces SPL$^{\text{LIFT}}$ and gives examples on what kinds of results are computed. Section 7.2 discusses expected differences in the results. Section 7.3 explains the design and implementation of the experiment in detail. Section 7.4 presents the results regarding performance and precision. Section 7.5 discusses the comparison results and lessons learned. Section 7.6 discusses related work on comparative studies while Section 7.7 rounds out the chapter with a conclusion and an outlook on future work.

7.1 Lifted Strategy and SPL$^{\text{LIFT}}$

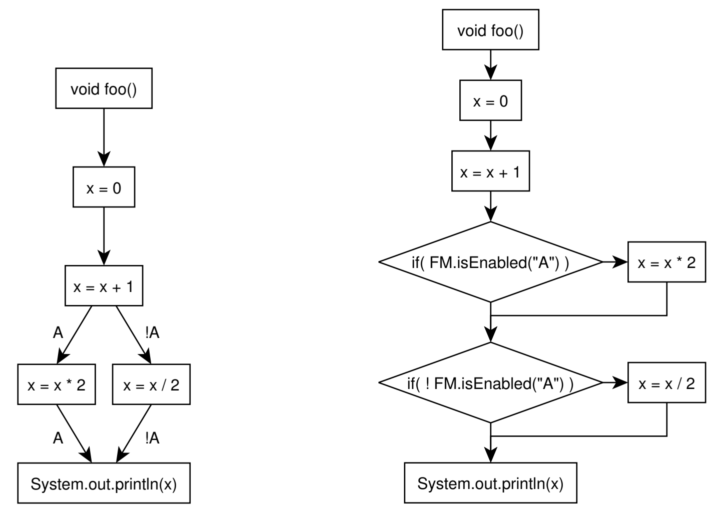
Recall variability-aware program analysis introduced in Section 2.4. This general concept for representing variability information in form of Boolean formulas is the foundation for making an analysis variability-aware, while not specifying a process for performing the analysis. The lifting framework presented by Brabrand et al. [Brabrand12] defines a systematic way for transforming a specific class of analyses into variability-aware analyses. SPL$^{\text{LIFT}}$ [Bodden13] implements Brabrand et al.'s lifting framework, which can automatically convert any interprocedural data-flow problem [Reps95] into a *lifted* version allowing to perform existing analyses in a variability-aware manner. The current implementation is based on Soot [Lam11], a program analysis framework for Java. Therefore, programs are transformed into Soot's intermediate representation Jimple before being analyzed. SPL$^{\text{LIFT}}$ uses the generic Heros solver [Bodden12] for IDE problems [Sagiv96]. The concrete analysis to be executed can be defined either by selecting an existing analysis implementation or by defining a new IFDS problem and implementing respective transfer functions. The analysis problem is then automatically *lifted* to a variability-aware SPL analysis.

The current implementation of SPL$^{\text{LIFT}}$ assumes that variability is defined using the Colored IDE (CIDE) [Kästner08]. In this tool colors represent features and variability is specified by coloring program statements. Source code may also be colored with multiple features, e.g., when defining feature interaction code. Listing 7.1 demonstrates how CIDE maps source code to features. The optional feature `A` is associated with the color yellow while the counter feature is associated with the color green. CIDE also provides a tool for generating product variants by selecting a set of features and composing code only for the selected features.

SPL$^{\text{LIFT}}$ retrieves the variability information (the colors) from CIDE, enumerates all features by assigning them a unique number, and converts this information into a bit set, with every bit uniquely representing a feature. The bit set is then attached to the corresponding Jimple statements used by the Soot analysis framework. A conjunctive feature interaction like $A \wedge B$ can be expressed by setting several bits in a bit set. In this way, every statement has an associated set of features. SPL$^{\text{LIFT}}$ then analyzes the annotated Jimple code and considers the variability information during the analysis by generating annotated results as outlined above for reaching definition sets.

7.2 Expected Analysis Differences

Due to the different way SPL$^{\text{LIFT}}$ and COACH! perform their analyses, COACH! is expected to be less precise as it needs one program encoding all possible product variants. This will usually lead to higher imprecision due to the over-approximations



(a) CFG for SPL$^{\text{LIFT}}$ **(b)** CFG for COACH!

Figure 7.1: Control flow graphs for lifted and delayed analyses.

```
1  public void foo() {
2    int x = 0;
3    x = x + 1;
4    if (FM.isEnabled("A")) {
5      x = x * 2;
6    }
7    if (!FM.isEnabled("A")) {
8      x = x / 2;
9    }
10   System.out.println(x);
11 }
```

Listing 7.1: SPL using load-time configuration options.

of the a feature-oblivious analysis which does not consider the dependencies between presence conditions. Consider the small SPL in Listing 7.1 and the resulting control flow graph in Figure 7.1b. There are four statements defining variable x, however, the print statement in the last line will never receive the value of statement in line 3, because one of the two assignment statements in line 5 and 8 will always be executed, thus overwriting the first assignment. As can be seen in Figure 7.1a, SPL$^{\text{LIFT}}$ is able to handle this situation precisely because it considers the presence dependency of these two assignments at the time of the data flow analysis. COACH!, on the other hand, first computes the reaching definitions completely variability-oblivious and also ignores the presence dependency. Therefore, it has to assume that every branch may be entered independently, which finally induces the spurious edge. Figure 7.2 shows the system dependence graph for the program in Listing 7.1. The highlighted edge will be computed by COACH! but not by SPL$^{\text{LIFT}}$.

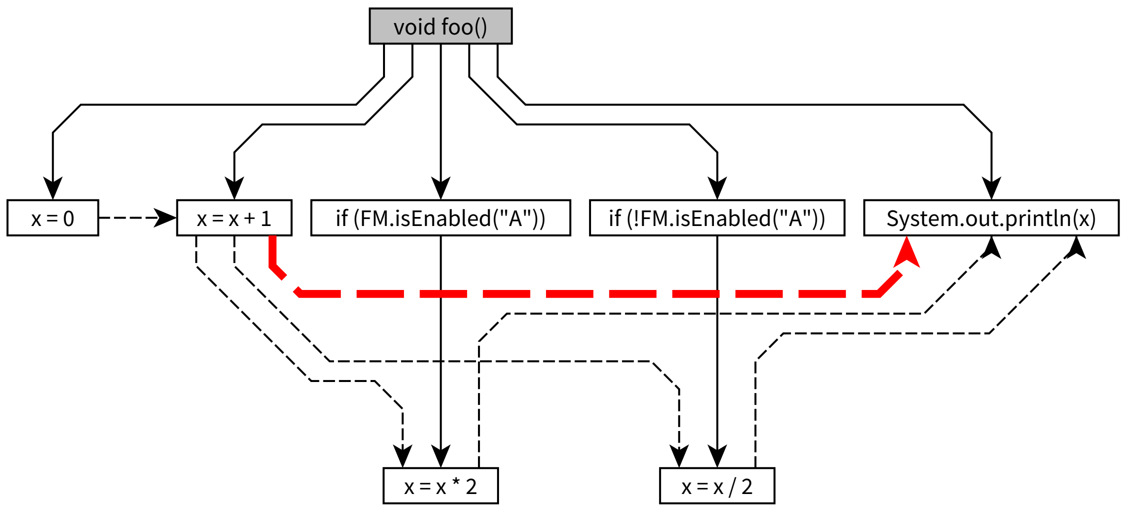


Figure 7.2: The variability-oblivious system dependence graph for the small product line. The highlighted edge is induced by COACH! due to an overapproximation but not by SPL$^{\text{LIFT}}$ since it can handle such situations precisely.

7.3 Experiment Design

Both the lifted and the delayed strategy aim at considering variability information in SPL analysis, but do it in different ways. This section describes the setup of our experiment comparing SPL$^{\text{LIFT}}$ and COACH! regarding performance and precision by applying the two approaches to five SPLs. Specifically, the experiment explores two research questions:

RQ1 – Performance. How do the approaches differ concerning the execution time required to compute the results?

RQ2 – Precision. How do the results differ regarding their precision?

In order to ensure a fair comparison of the analysis results and the measured run time, tool chains for SPL$^{\text{LIFT}}$ and COACH! are prepared to process the same inputs and produce the same output data structures. A prerequisite for the comparison is that both approaches compute the same SDGs if variability is ignored. This must be the case because both tools use the IFDS framework to compute the data dependencies. Both tools have therefore analyzed each SPL while completely disabling the extraction of feature information. The experiment continued only after checking that the pairwise results of both tools were equal.

Figure 7.3 shows an overview of the two-phase experiment. In the *preparation phase*, the source code and feature information of the chosen SPLs are parsed. Specifically, a CIDE project is parsed and transformed into Soot's intermediate representation Jimple. The colors representing features are extracted from the CIDE project. The Jimple statements are annotated with feature tags associating them with a set of features. If

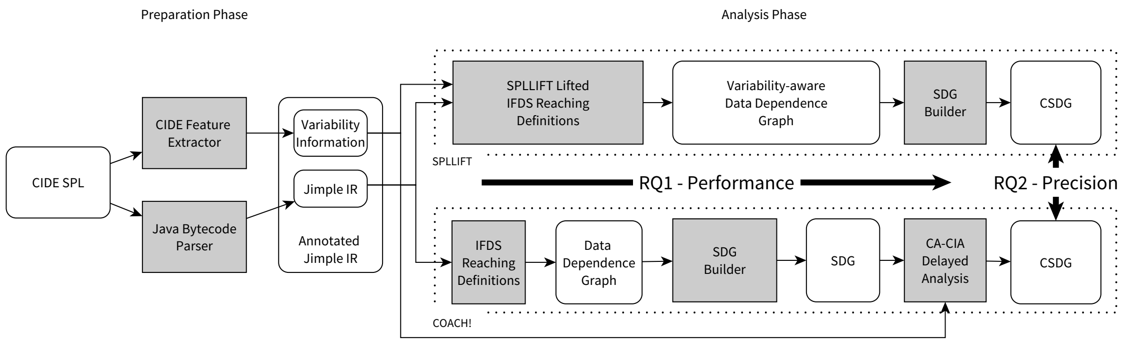


Figure 7.3: An overview of the conducted experiment. In the preparation phase, the source code of the SPLs is parsed ignoring variability. SPL$^{\text{LIFT}}$ and COACH! are triggered to build the CSDG and the resulting CSDGs are compared and expected to be equal. In the analysis phase, the source code and the variability information of the SPLs are parsed. SPL$^{\text{LIFT}}$ and COACH! are triggered to build the CSDG. The time required to build the CSDG is measured and the precision of the variability information is compared.

a Jimple statement does not have a feature tag, it is assumed that the statement is included in all possible product variants.

In the *analysis phase* the two alternative approaches are executed: the SPL$^{\text{LIFT}}$ tool chain indicated by the upper dotted box and the COACH! tool chain indicated by the lower dotted box. Starting at this point, SPL$^{\text{LIFT}}$ and COACH! are triggered to build the CSDG. The tool chain thereby measures and compares the time required to build the CSDGs and the precision of the variability information in the resulting CSDGs.

7.3.1 Adapting the Tool Chains

In order to ensure that both tools can work with the selected SPLs and produce comparable results, it was necessary to adapt the tool chains.

For COACH!, we added a preprocessing step for adequately extracting the variability information from CIDE product lines. The COACH! approach has been designed to handle load-time configuration options and it can therefore not directly analyze CIDE product lines. As described in Section 3.2, the options were identified in a separate step and corresponding seed conditions were added directly to the SDG. In other words, the approach assumes that variability is implemented using branch statements in the programming language. Further, COACH! assumes that variability is on the level of basic blocks and that there is one preceding block that controls the variable block. Hence, COACH! was modified such that variability can be handled at the level of statements. Therefore, it stores seeds conditions on control dependence edges and not on SDG nodes as before. In this way, COACH! does not need any kind of variability encoding that would alter the program code and possibly affect the analysis tools' behavior.

Further, in the tool chain for SPL$^{\text{LIFT}}$, a postprocessing step to build the CSDG was introduced after the lifted analysis step. This CSDG builder is exactly the same as for COACH!, except that it uses the results of the lifted IFDS reaching definitions analysis to create data dependence edges. However, to ensure a fair comparison, the steps for generating the Jimple IR, for extracting variability information, and for running the lifted analysis remained unchanged.

7.3.2 Evaluating Performance

RQ1 investigates the run-time performance of the two tool chains. The performance test drivers repeatedly execute the two tool chains to measure the time required for the analysis, which is the focus of the experiment. For the SPL$^{\text{LIFT}}$, the measured time includes the time required to solve the lifted reaching definitions problem and the time required to finally build the CSDG. This corresponds to the steps *SPLLIFT Lifted IFDS Reaching Definitions* and *SDG Builder* in Figure 7.3. For the COACH!, the measured time includes the time required to do the variability-oblivious reaching definitions analysis, building the SDG and propagating the variability information in the SDG. This corresponds to steps *IFDS Reaching Definitions*, *SDG Builder*, and *CA-CIA Delayed Analysis* in Figure 7.3. However, measurements ignore the time required to parse the product lines' source code and to extract the feature information as these steps were exactly the same for both tool chains.

All measurements have been conducted on an Intel Core i7-4770, 3.40 GHz, 16 GB DDR3-RAM machine running MS Windows 7 64-bit. Since the Java VM compiles just-in-time, some overhead might arise due to so-called warm-up effects of the JIT compiler and other VM internal processes. The measurements therefore follow a practice which is commonly used in performance engineering [Hofer16]. The test driver executes the tools in so-called *rounds*, each consisting of 50 iterations. An iteration is one run of one tool chain. The complete Soot environment and any caches of the whole analysis tool chains were completely reset between the iterations. Furthermore, every round has been started in a fresh Java VM process to also reset VM internal caches. The run-time figures are then plotted on a graph to be able to visually identify the warm-up phase of the Java VM. The number of iterations in the warm-up phase are dependent on the analyzed product line and lies around 20 iterations. The warm-up iterations were then excluded from any measurements.

The final value for the run time is computed by first computing the median for every round and further computing the geometric mean over all rounds. In this way the influence of VM warm-up effects could be kept very low in the measurements.

7.3.3 Comparing Precision

RQ2 compares the presence conditions in the two computed CSDGs. The test driver stored the CDSGs' data dependence edges and their presence conditions in a file, grouping the edges by the method they belong to. Specifically, it dumped the CSDG into a result file by iterating over all reachable method nodes, writing a header for every method node and dumping the successor edges of all nodes within a method. The list of methods and the list of edges per method were then sorted to achieve result files which are easily comparable. The dump file also stores the condition representing the feature model as proposed by Batory [Batory05].

The two files generated by the test driver for the SPL$^{\text{LIFT}}$ and COACH! tool chains are then compared to compute following metrics:

- $E_{SPLLIFT}$ and E_{COACH} represent the **number of data dependence edges** in the SPL$^{\text{LIFT}}$ and COACH! result files.
- EIC is the number of **equal edges ignoring conditions**, i.e., edges with the same source and destination in both result files, regardless of the attached conditions.
- EqE represents the number of **equal edges** that have equal conditions in both result files.
- $ACP_{SPLLIFT}$ and ACP_{COACH} are the **condition precision** metrics of the comparable edges in the SPL$^{\text{LIFT}}$ and COACH! result files, i.e., edges which have same source and destination. The metrics thus quantify the precision of the presence conditions in the CSDG. As a presence condition represents a set of valid product configurations, the precision of a presence condition depends on the number of possible product variants it represents, i.e., fewer product variants mean higher precision. For example, the presence condition `true` stands for any possible product variant and means that the annotated element may occur in any possible product. Hence, the simplest but most imprecise variability information is the condition `true`.

Since BDDs are used to represent presence conditions, it is easily possible to compute all satisfiable assignments of a presence condition and take this as a measurement for the possible product variants the condition represents. However, conditions do not necessarily contain every feature and the dependencies in the feature model may further exclude product variants. It is therefore necessary to take the condition representing the feature model and combine it with the presence conditions using the conjunctive operator `AND`.

Based on those considerations, the following formula defines the *precision* of a presence condition c:

$$prec(c, fmcond) = 1 - \frac{count(allsat(c \wedge fmcond))}{count(allsat(fmcond))}$$

Project	Size [LOC]	Features
berkeleyDB	84,000	56
gpl	1,400	29
lampiro	45,000	20
mm08	5,700	34
prevayler	80,000	5

Table 7.1: SPLs used in the experiment.

where $fmcond$ is the formula representing the feature model. That means, the precision of a presence condition is defined by 1 minus the fraction of the number of product variants possible when considering the presence condition c and the feature model $fmcond$ and the number of all product variants possible in $fmcond$. For illustration, assume there is feature model with two optional features f_0 and f_1 resulting in the feature model condition $f_0 \vee f_1$. Let's further assume that there is no feature interaction, then there will be code that is annotated with f_0 or with f_1 only. All satisfiable assignments of condition f_0 are $allsat(f_0) = \{f_0 = true\}$, i.e., there is only one satisfying assignment. However, also considering the condition representing the feature model, the formula for all possible assignments is $f_0 \wedge (f_0 \vee f_1)$ and there are two possible satisfiable assignments, namely $\{(f_0 = true, f_1 = false), (f_0 = true, f_1 = true)\}$. Hence, $count(allsat(f_0 \wedge (f_0 \vee f_1)))$ is 2. Then the number of all possible product variants is determined by $count(allsat(f_0 \vee f_1))$ which equals to 3. Thus, the precision of the presence condition f_0 in feature model $f_0 \vee f_1$ is $1 - \frac{2}{3} = \frac{1}{3}$.

7.3.4 Selected SPLs

The five SPLs *BerkeleyDB*, *Graph PL (GPL)*, *Lampiro*, *Mobile Media 08 (MM08)*, and *Prevayler* are the input for the experiment. Four of these SPLs were chosen, as they were already used in an earlier evaluation of SPL$^{\text{LIFT}}$ [Bodden13] and variability information is available in the CIDE format. The *Prevayler* product line is an additional evaluation case. Table 7.1 summarizes the lines of code and number of features for these projects. The following briefly describes the projects:

BerkeleyDB[1] is a high-performance data base system for storing key-value data. It was originally not designed as an SPL and had been written in C without explicit feature information. However, the Java version of the BerkeleyDB was decomposed into 56 features by Kästner et al. [Kästner07].

The *Graph Product Line* is a collection of graph algorithms design as an SPL [Kästner08]. It has just about 1,400 LOC and 29 features, which are heavily used.

[1]`http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`

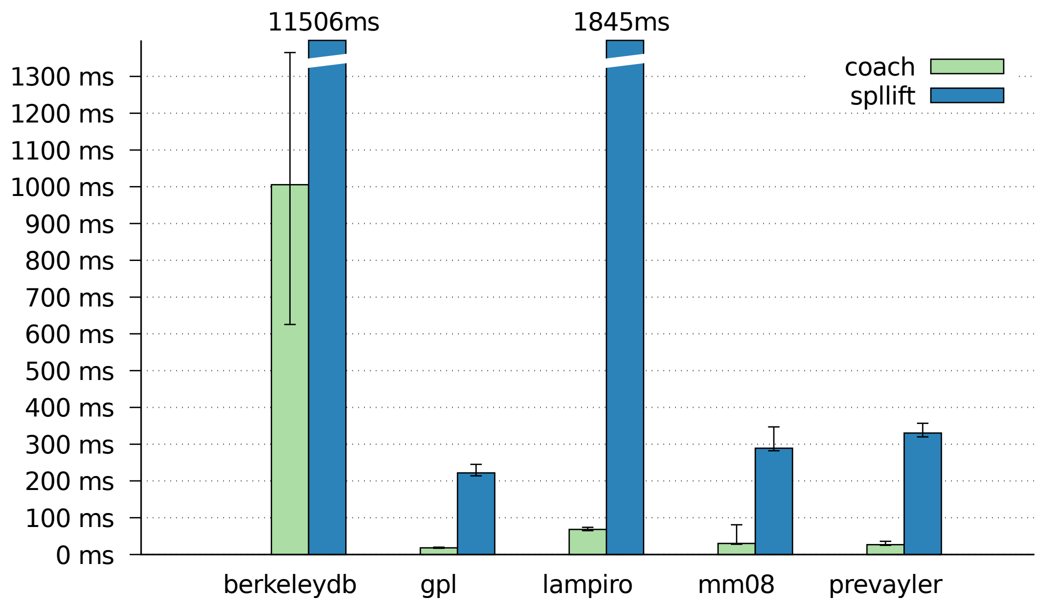


Figure 7.4: Results for **RQ1** on run-time performance of SPL$^{\text{LIFT}}$ and COACH!.

Lampiro is an open source implementation of an XMPP instant messenger. It is a J2ME application with about 45,000 LOC and 20 features and has been designed as SPL from its inception.

Mobile Media 08 is an academic project and has been designed as SPL from its inception. It is a rather small J2ME application with 5,700 LOC and 34 features.

Prevayler is an in-memory data base system for Java. It was not designed as SPL but researchers have annotated its variability. Despite the size of 80,000 LOC the product line has only 5 features.

The source code of these projects is available on the website SPL2go[2]. However, the projects used for this experiment are variants with slight modifications as they are used in [Bodden13]. In particular, the projects have an additional class `DummyMain`, to ensures that code is reachable as the Soot analysis would ignore unused code.

7.4 Experiment Results

The following sections report comparative results regarding the performance and precision of SPL$^{\text{LIFT}}$ and COACH!.

[2]`http://spl2go.cs.ovgu.de/`

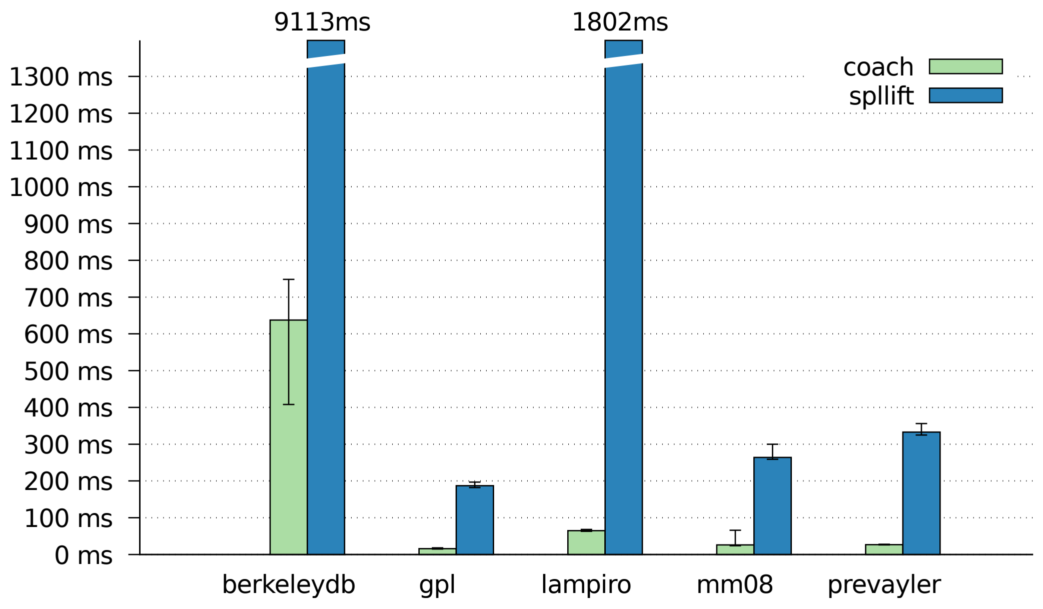


Figure 7.5: Results for the smoke test on run-time performance of SPL^{LIFT} and COACH!. During the smoke test, both analysis tools completely ignored variability.

7.4.1 RQ1 – Performance

The results of the performance evaluation are shown in Figure 7.4. Additionally, Figure 7.5 shows the results of the smoke test, i.e., the tools completely ignored variability during analysis. The values shown represent the geometric mean over all rounds. The median of all iterations is computed for each round excluding identified warm-up iterations (cf. Section 7.3). Furthermore, the diagram also shows error bars providing information about the deviation of the single iterations. The error bars are very small, indicating that the measured run times for the single iterations are stable and trustworthy. There are no error bars for SPL^{LIFT}'s value for *BerkeleyDB* and *Lampiro* because the values are out of the diagram's range and error bars would not make sense in this case. But since the value for COACH! is about 10 times smaller, this does not matter anyway.

The performance results answering RQ1 show that building the CSDG using COACH! is significantly faster compared to SPL^{LIFT}. We expected that the step *SDG Builder* would take longer in the SPL^{LIFT} tool chain as accessing the variability-aware reaching definitions results is more time consuming if additionally storing a presence condition.

Overall, comparing the total run time (cf. Figure 7.4) shows that the COACH! tool chain runs notably faster. Explaining this significant difference is difficult and providing a definite answer is not possible. The discussion section will report some possible explanation.

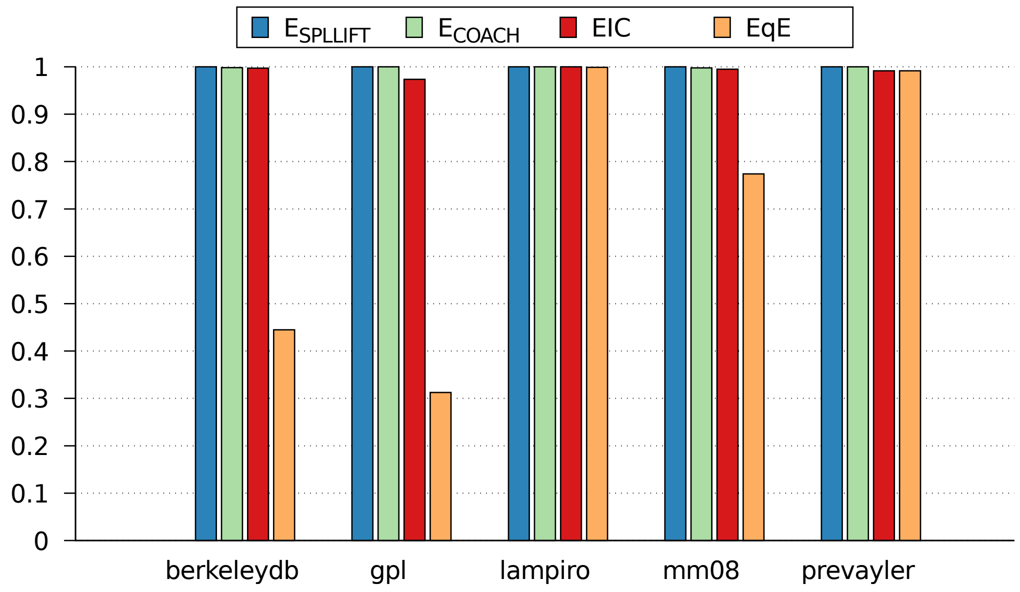


Figure 7.6: Results for **RQ2** comparing the precision of the analyses. The number of edges is normalized to SPL$^{\text{LIFT}}$'s results to ease comparison.

7.4.2 RQ2 – Precision

The results shown in Figure 7.6 compare the results regarding precision. The diagram shows the results for metrics $E_{SPLLIFT}$, E_{COACH}, EIC and EqE for each product line, normalized to $E_{SPLLIFT}$.

Comparing the bars for metrics $E_{SPLLIFT}$ and E_{COACH} show that COACH! does not produce more data dependence edges, i.e., the number of edges is almost equal. Note, the results only contain edges with a condition other than `false`. Therefore, if an edge is missing in the result file of SPL$^{\text{LIFT}}$ but the edge is present in the result file of COACH!, we know that the missing edge could be excluded due to a more precise computation (cf. Section 7.2). The metric EIC measures the intersection of edges between the two tools, i.e., a higher value is better. For example, the value of EIC for GPL is nearly 1.0, i.e., almost all data dependence edges computed by SPL$^{\text{LIFT}}$ were found by COACH!. This metric can be taken as an argument for soundness of the comparison because the structure of the SDG will therefore be very similar. Metric EqE is a fraction of EIC and indicates how many edges also have the same condition. Again, a higher value is better but EqE can never be larger than EIC.

Figure 7.7 shows the precision for presence conditions (cf. metrics $ACP_{SPLLIFT}$ and ACP_{COACH}) using box plots. We computed the precision for all conditions of pairs of edges with the same source and destination but with a different condition. That means, all edges which are not within EqE are considered. For example, if SPL$^{\text{LIFT}}$ outputs the edges $\{a \xrightarrow{c_0} b, a \xrightarrow{c_1} c\}$ and COACH! outputs the edges $\{a \xrightarrow{c_2} b, a \xrightarrow{c_3} c\}$ the

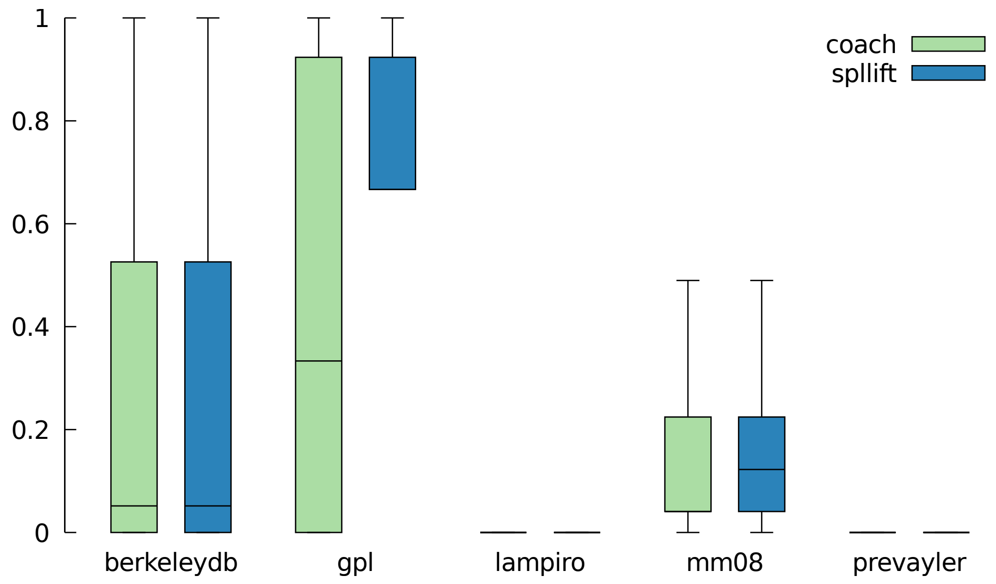


Figure 7.7: RQ2. The distribution of the precision of presence conditions of edges with the same source and destination but with a different conditions. Lampiro and Prevayler don't have any such edges with different conditions.

precision value is computed for c_0, c_1, c_2 and c_3. The boxes in the Figure 7.7 show the 2nd and 3rd quartile of the distribution of the $ACP_{SPLLIFT}$ and ACP_{COACH} values. The whiskers cover values from the 12.5% to the 87.5% quantile. As the box plots show, both tools produce conditions covering the full precision range from 0.0 to 1.0. The precision of the conditions on edges with the same source and destination is on average notably worse for COACH! than for SPL^{LIFT} in the case of GPL. The two tool chains do not compute any different conditions for the product lines Lampiro and Prevayler while SPL^{LIFT} slightly outperforms COACH! in case of MM08 because the median is lower for COACH!.

7.4.3 Threats to Validity

The main threats to validity in this work are related to performance measurement and the product line selection.

Performance measurements. Since the whole tool chain is written in Java, the measured run time also contains time consumed by internal Java VM subsystems and the measurements may slightly vary from run to run. However, the applied measurement procedure developed in the performance engineering community allows to report reliable results. For instance, repeating all measurements 50 times in the same VM process and not considering outliers caused by warm-up effects of the VM. For this reason, we also report the median of all run time values.

The performance numbers for SPL^{LIFT} are significantly better than the results presented in the original publication [Bodden13], i.e., SPL^{LIFT} performed a lot better in this experiment setting although the analysis part of SPL^{LIFT} was not modified and the original snapshot provided by one of the authors of [Bodden13] were used. This was somehow surprising as one would expected the number of the run-time performance measurements for reaching definitions to be similar when applying the approach to the same SPLs. A reasonable explanation for the difference is that the measurements in the original paper have been made about four years earlier and better hardware and in particular optimizations in the Java VM have caused the improvements.

Selected product lines. There is a potential bias caused by the selection of the analyzed product lines. Our aim was to run SPL^{LIFT} in the very same way as described by Bodden et al. [Bodden13] but just for the reaching definitions analysis. Therefore, this experiment replicated their analyses using the same product lines (see Section 7.3.4), allowing to argue that the results computed by SPL^{LIFT} are correct.

7.5 Discussion

Performance The performance measurements show, that the delayed analysis is notably faster than the lifted analysis when building the CSDG. There are two main reasons for this difference. First, as explained in Section 7.2, the delayed approach uses variability-oblivious algorithms for doing control and data flow analysis. Computing the control and data flow dependencies is, in the first place, faster because they do not have to consider variability. However, in doing so they sacrifice precision for performance. Then some additional effort is required for recovering the variability information. However, this study showed that the recovery was fast in the case of building the CSDG. Second, the delayed variability analysis approach allows optimizing recovery of variability information for a specific purpose. For instance, COACH! was optimized for the problem of adding variability information to control and data flow dependencies in the CSDG.

Applicability The lifted variability analysis and in particular the SPL^{LIFT} tool are more general as they are designed to handle any data flow analysis which can be formulated as IFDS problem. On the other hand, a delayed approach is usually designed for a specific problem. Recall that the delayed approach first does a variability-oblivious analysis and then recovers variability information for a specific purpose, e.g., variability information in the SDG as in the COACH! approach.

Another restriction for the delayed strategy is that it requires a source code base that can be compiled without preprocessing while lifted strategy is able to handle preprocessor-based product families. On the other hand, COACH! is built for handling load-time variability, a type of variability SPL$^{\text{LIFT}}$ cannot handle currently. However, this difference is basically just due to the different goals and it has been shown that it is possible to transform programs to use another variability mechanism [vonRhein16].

Precision A higher precision is expected for SPL$^{\text{LIFT}}$ as it can handle certain situations more precisely, as already pointed out in Section 7.2. The results show that for product lines *GPL*, *Lampiro*, *MM08*, and *Prevayler* the data dependence edges found by both tools when not considering conditions are almost equal (cf. metric EIC). That means the differences are quite small and both approaches produce very similar output which does not confirm the expectation in the first place. However, looking at the conditions of edges reveals that SPL$^{\text{LIFT}}$ is more precise than COACH! for systems with high variability, like *GPL* (cf. metrics EqE, $ACP_{SPLLIFT}$, and ACP_{COACH}). The results show that SPL$^{\text{LIFT}}$ is more precise compared to COACH! for systems with high variability. For example, *BerkeleyDB* is a quite large software system having just a little variability whereas *GPL* is a small and highly configurable software.

Result Implications The decision for a specific analysis approach will first be determined by the kind of the employed variability mechanism. When setting up the tool chain for the experiment it came out that using COACH! for compile-time variability was not straightforward and required some time-consuming adaptations. On the other hand, using SPL$^{\text{LIFT}}$ for load-time variability is also not supported out of the box. Second, the choice also depends on the kind of problem to be solved. SPL$^{\text{LIFT}}$ supports every IFDS problem without any further effort, however, the bare results provided by SPL$^{\text{LIFT}}$ will usually require some postprocessing. For instance, in this case the results of SPL$^{\text{LIFT}}$ are provided via an interface allowing to query data dependencies between statements. The postprocessing step then was to use this interface for building the CSDG, while COACH! directly works on the final data structure, e.g., the SDG.

7.6 Related Work

Related work for variability-aware program analysis and tracking of load-time configuration options has already been discussed before (e.g. for Chapter 5). This section therefore concentrates on related work doing some kind of comparison between analysis techniques.

For comparing different implementation strategies, Brabrand et al. [Brabrand12] have implemented four different versions of their lifting framework. Analysis $A1$ (brute force) builds all possible product variants and analyzes them one by one using a standard data flow analysis. Analysis $A2$ (consecutive feature-sensitive analysis) is a variability-aware analysis, but analyzes just one product configuration at a time. The simultaneous feature-sensitive analysis $A3$ uses a lifted lattice analyzing all possible product variants at once by maintaining one lattice element per valid product variant. The shared simultaneous feature-sensitive analysis *A*4 further improves *A*3 by sharing lattices where possible (cf. late splitting and early joining in Section 2.4). The authors compare the run-time performance of all four analysis variants among each other and therefore do a very similar comparison as in this experiment. However, this experiment compares two variability-aware approaches and since the precision of the results are considered additionally.

Liebig et al. [Liebig13] compare their analyses to sampling-based approaches, which use specific strategies to generate concrete product variants that are analyzed. The used strategies aim at producing samples that are representatives for certain groups of variants and therefore do not cover all product variants. They showed that variability-aware analyses are faster and have full variant coverage compared to sampling strategies. Again, the paper does not compare the quality of results which are assumed to be deterministic and equal.

Lillack et al. [Lillack14] do not compare their approach to others but they compare the results of their tool to an *oracle* and measured the accuracy of the results. This comparison is similar to the precision comparison in this experiment. However, the evaluation in this chapter does not use a defined ground truth but uses defined metrics to compare the precision of two strategies.

7.7 Summary

This chapter reported results of a comparative study of two different techniques for providing variability-awareness in program analysis. The lifted strategy implemented in the SPL$^{\text{LIFT}}$ tool considers variability already during the parsing stage. Therefore, the results of SPL$^{\text{LIFT}}$'s analysis are as precise as analyzing all possible product variants separately. However, SPL$^{\text{LIFT}}$ targets compile-time variability and cannot handle the propagation of load-time configuration values. The delayed strategy implemented by the COACH! tool presented in this thesis recovers variability only when needed. This enables COACH! to handle load-time configuration but it cannot handle compile-time variability if the structure of the program varies.

CIDE is an development environment for SPLs and allows to specify variability by coloring statements. CIDE does not allow to break the structure of the program but it still uses compile-time variability and therefore CIDE projects are suitable input for both approaches.

The study described in this chapter compared the tools COACH! and SPL$^{\text{LIFT}}$ regarding performance and precision using CIDE projects as input. The chapter also explained the expected differences in performance and precision due to the different analysis strategies of the two tools.

The results show that the delayed strategy is significantly faster (about 10x) for all input SPLs but generally less precise. However, the degree of imprecision depends on the variability complexity of the particular SPL. Furthermore, implications of the results are discussed that may help researchers and practitioners that need to perform variability-aware analysis in their context.

Chapter 8

Conclusions

The motivation for the research presented in this thesis was twofold: (i) variable software systems, and (ii) multi-language software systems. It particularly addressed the problems arising in program analysis from load-time configuration options during maintenance of variable software and the use of multiple programming languages.

A main goal of the methods presented in this thesis was to support developers during maintenance and evolution of such software systems. Therefore, we defined three research goals: **RG1** support for assessing the impact of changes in variable software, **RG2** methods for creating views with reduced source code complexity during maintenance, and **RG3** program analysis support for multi-language software systems. These research goals call for variability-aware and cross-language analysis support.

In distinction to other approaches, the methods of the thesis pursue a delayed variability analysis approach which especially target on handling load-time configuration options. The CSDG has been introduced as a variational data structure for supporting the analysis methods. It is a system dependence graph representing all control and data dependencies globally in a program together with the variability of the system in the form of presence conditions. The CSDG is computed based on the delayed variability analysis approach as first a variability-oblivious SDG is built and variability is added to the SDG in the last stage. The CSDG is the basis for the ICD approach to determine code that will never be executed in a concrete product configuration and the CA-CIA approach for doing a slicing-based CIA which also considers the variability of systems.

In the following the contributions of this thesis are summarized:

Contribution 1 The conditional system dependence graph (CSDG) presented in Chapter 3 is the basic data structure for the configuration-aware program analysis methods. It is based on system dependence graph (SDG) and is built from the SDG by annotating edges with presence conditions. The chapter also showed how variability mechanisms, i.e., the implementation of variability in the source code, are abstracted for extracting presence conditions. Further, it showed how to deal with non-Boolean configuration options and how to deal with situations where configuration variables are set together with normal program variables. The performance evaluation showed that building the CSDG is fast and in our case scaled with the size of the analyzed software.

Contribution 2 Chapter 4 introduced the *inactive code detection (ICD)* approach for finding code in product variants that cannot be executed because of the configuration. The inactive code is dead code in a certain product configuration but may be used in other configurations. The technique allows hiding inactive code in a product configuration by performing a reachability analysis on the CSDG. Furthermore, we also showed how to use the ICD approach together with ECCO for recovering feature-to-code mappings. ECCO is an approach that allows recording variability traces in clone-and-own product lines and relies on differences in the source code of variants. However, with load-time configuration options, there are no source code differences in products but only differences in active and inactive code and therefore the inactive code must first be identified. The evaluation showed that the ICD approach is effective and accurate. We also showed that the combination of ICD and ECCO produces sound results with respect to the input variants, i.e., it was possible to re-compose the input product variants from the extracted traces.

Contribution 3 Chapter 5 introduced the configuration-aware change impact analysis (CA-CIA) approach. CA-CIA uses the delayed variability concept to make program slicing based CIA configuration-aware using the CSDG. It propagates presence conditions representing the variability of the analyzed software system. CA-CIA avoids global propagation of variability information within the whole CSDG by determining possibly influencing statements. The evaluation of CA-CIA showed that the approach is beneficial since it provides variability information and can reduce change impact size because of configuration. The run-time performance showed that the technique scales for industrial systems.

Contribution 4 Chapter 6 presented the CAM-CIA approach for performing compositional CA-CIA using program dependence graphs as modules. It has been shown that the CA-CIA can be performed within a module without restrictions, resulting in sets of nodes and edges where the edge conditions may contain placeholders. CAM-CIA also described how to compose analysis modules by resolving placeholder variables. The evaluation showed the correctness of the CAM-CIA approach and its benefits compared to the non-modular CA-CIA approach.

Contribution 5 Chapter 7 reported results of a comparative study of two different techniques for providing variability-awareness in program analysis. The lifted strategy implemented in the SPL$^{\text{LIFT}}$ tool considers variability already during the parsing stage. The delayed strategy implemented in the COACH! tool presented in this thesis recovers variability only when needed. The study described in this chapter compared the two tools regarding performance and precision using CIDE projects as input. The results show that COACH! is significantly faster (about 10x) but generally less precise.

List of Figures

List of Tables

Curriculum Vitae

Personal Information

Name	Dipl.-Ing. Florian Friedrich Angerer, BSc
Date of Birth	July 22, 1986
Personal Status	married, two children
Nationality	Austrian
Email	florianangerer@gmx.at

Education

2012 - 2017	PhD in Computer Science – Johannes Kepler University Linz
Autumn 2010	Semester Abroad – Swiss Federal Institute of Technology
2010 - 2012	MSc in Computer Science (with distinction) Johannes Kepler University Linz
2008, 2009, 2010	Merit Scholarship
2007 - 2010	BSc in Computer Science (with distinction) Johannes Kepler University Linz

Employment History

Feb 2013 - Jan 2017	CDL MEVSS - Johannes Kepler University *Researcher / Software Engineer* My research focused on program analysis for configurable and modular software systems to support evolution and maintenance. Besides the scientific research, I was responsible for the development of the analysis framework implementing my approach. This involved supervision of a small team with up to three student researchers working in this project.

Mar 2012 - Jan 2013	Johannes Kepler University *Researcher / Software Engineer* Conducted a project in cooperation with Manus.M GmbH with the goal of automated network configuration for mobile devices.
Mar 2011 - Jan 2013	Johannes Kepler University *Researcher / Software Engineer* In cooperation with ENGEL Austria GmbH, I developed an analysis framework for analyzing PLC programs for code smells and defects. The tool is currently used on ENGEL's nightly build server for their PLC software.
2006 - 2007	Arbeiter Samariterbund OÖ *Community Service*

Experience

Conference/Workshop Talks

ICSME 2016 (Raleigh, NC, USA)
ASE 2015 (Lincoln, NE, USA)
FOSD Meeting 2015 (Traunkirchen, Austria)
ICSME 2014 (Victoria, BC, Canada)
SPLC 2014 (Florence, Italy)
ASE 2014 (Vasteras, Sweden)
ETFA 2013 (Cagliari, Italia)
FOSD Treffen 2013 (Dagstuhl, Germany)
ETFA 2012 (Cracow, Poland)

Teaching

Autumn 2013 Grundlagen der Programmierung (Introduction to Programming)
Spring 2014 Praktische Informatik 2 (Data Structures)
Spring 2015 Praktische Informatik 2 (Data Structures)

Supervision of Master's Thesis

Analysis of PLC Programs Using the Program Analysis Framework SOOT (DI Andreas Grimmer)

Supervision of Bachelor's Thesis

Jimple Code Generator for SFCs (DI David Auinger)
IEC Editor in Eclipse (Alois Mühleder)

Glossary

AE application engineer. 5, 7, 119, *Glossary:* application engineer

AppCo Application Composer. 4, 47, 52, 119

application engineer A software engineer realizing a concrete application satisfying the specific application requirements by exploiting the commonality and variability of the SPL.. 5, 119

AST A tree structure representing syntactic structure of the program's source code. Each node corresponds to some syntactic element of the code.. 8, 119

AST abstract syntax tree. 8, 21, 29, 33, 34, 40, 45, 54, 119, *Glossary:* AST

Automation Platform KEBA's basic platform for creating domain-specific solutions. It provides real-time capabilities, communication services, debugging and logging facilities, runtime systems for automation-specific languages, et cetera.. 4, 5, 119

BDD binary decision diagram. 28, 101, 119

C preprocessor Language facility performing a first compilation step [Kernighan88]. It can include files, replace tokens, or conditionally include source code.. 13, 119

CA-CIA configuration-aware change impact analysis. 9, 10, 19, 61, 67–70, 72, 76–78, 80, 82, 84–88, 90, 91, 93, 111–113, 119, *Glossary:* configuration-aware change impact analysis

CAM-CIA configuration-aware modular CIA. 9, 10, 87, 89–91, 93, 112, 114, 119, *Glossary:* configuration-aware modular CIA

CDS control dependence subgraph. 84, 119, *Glossary:* control dependence subgraph

CFG control flow graph. 17, 20, 32, 76, 119, *Glossary:* control flow graph

change impact analysis The process of determining the potential effects of a proposed modification in the software [Bohner02].. 9, 17, 61, 119

CIA change impact analysis. 9, 17, 18, 57, 61, 63, 68, 86, 88, 91, 111, 112, 119, *Glossary:* change impact analysis

CIDE Colored IDE. 58, 109, 112, 119

clone-and-own Ad-hoc approach for reusing software by simply cloning existing implementations and then adapting the clone to fulfill new requirements.. 119

COACH! configuration aware change !mpact. 10, 76–78, 83, 95–101, 103–109, 112, 114, 119

conditional system dependence graph Refer to Chapter 3.. 7, 23, 35, 111, 119

configuration option A variable that can be specified externally from the program and thus might have influence on the program's behavior.. 14, 119

configuration-aware change impact analysis Refer to Chapter 5.. 9, 61, 86, 112, 119

configuration-aware modular CIA Refer to Chapter 6.. 9, 119

control dependence subgraph The subgraph of the PDG just containing control dependencies.. 84, 119

control flow graph A directed graph structure encoding all possible program execution paths. The graph consists of an entry, exit, and basic block. 17, 32, 119

CPP C preprocessor. 13, 14, 119, *Glossary:* C preprocessor
CSDG conditional system dependence graph. 7–11, 21–23, 26, 27, 29, 31–36, 39–42, 44, 57, 58, 61, 63, 65–72, 74, 76, 82, 86–89, 91, 100, 101, 108, 111–114, 119, *Glossary:* conditional system dependence graph

data dependence graph The subgraph of the PDG just containing data dependencies.. 19, 119
DDG data dependence graph. 19, 119, *Glossary:* data dependence graph
delayed variability Performing an analysis completely variability-oblivious, i.e., ignoring any variability in the first place, and then recovering the variability and augmenting the results. 3, 21, 22, 61, 67, 84, 85, 95, 111, 113, 119

early joining 20, 21, 119
ECCO extraction and composition for clown-and-own. 51–54, 59, 112, 119

HMI human-machine interface. 119, *Glossary:* human-machine interface
human-machine interface The part of the application, that handles the human-machine interaction.. 119

ICD inactive code detection. 8–10, 38, 39, 45, 47, 50, 52, 54, 58, 59, 61, 111–113, 119, *Glossary:* inactive code detection
ICFG interprocedural control flow graph. 119, *Glossary:* interprocedural control flow graph
IEC 61131-3 Is the third part of the IEC 61131 standard for programmable logic controllers. Part 3 defines four programming languages for PLC programming. 4, 7, 8, 33, 34, 36, 76, 119
IFDS inter-procedural, finite, distributive, subset problems [Reps95]. 119
inactive code detection Refer to Chapter 4.. 8, 38, 58, 112, 119
interprocedural control flow graph A control flow graph where all call and reutnr nodes are connected to the corresponding entry and exit nodes of the single control flow graphs of procedures.. 119

Jimple Soot's intermediate representation for program code. 11, 33, 34, 119

KConfig KConfig is a configuration database organizing configuration options for the Linux kernel in a tree structure.. 56, 119
KEBA KEBA. 4–7, 33, 34, 42, 45, 47, 48, 52, 61, 77, 86, 91, 113, 119
KEBA AG KEBA AG (`http://www.keba.com`). v, 4, 42, 119
KePlast KEBA's domain-specific solution platform for plastics injection molding machines.. 4, 7–10, 33, 34, 38, 42–46, 48, 50, 52–56, 58, 76, 77, 91, 93, 113, 119

late splitting 20, 21, 119
lifted analysis A program analysis converted to be able to analyze a whole product family. 21, 95, 119
liveness analysis A classical data-flow analysis computing for each program point which variable values are still required.. 119
load-time configuration option A configuration option which value is bound during the initialization phase of a program.. 14, 15, 27, 111, 119

MEVSS Monitoring and Evolution of Very-Large-Scale Software Systems. 119

PDG procedure dependence graph. 17, 34, 87, 88, 90, 119, *Glossary:* program dependence graph

PLC programmable logic controller. 33, 119, *Glossary:* programmable logic controller

program dependence graph Explicitly represents data and control dependencies of a procedure [Ferrante87].. 17, 119

programmable logic controller A hard real-time system usually controlling manufacturing processes.. 33, 119

SDG system dependence graph. 7, 8, 17, 23, 24, 26, 27, 29, 31, 32, 34–37, 41, 50, 63, 69, 76, 89–91, 98, 100, 105, 108, 111, 113, 119, *Glossary:* system dependence graph

software product line A set of software-intensive systems sharing common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [Clements02].. 1, 15, 119

Soot A Java analysis and optimization framework `https://sable.github.io/soot/`. 33, 34, 36, 103, 119

SPL software product line. 1–3, 9, 10, 15, 16, 19–21, 48, 50, 53, 59, 95, 99, 107, 109, 110, 113, 119, *Glossary:* software product line

SPL$^{\text{CIA}}$ Lifted change impact analysis tool using SPL$^{\text{LIFT}}$. 119

SPL$^{\text{LIFT}}$ A tool for performing data flow analyses formulated in the IFDS framework for SPLs.. viii, 10, 95–109, 112, 114, 119

system dependence graph A directed graph representing different kinds of dependencies between program elements in the whole program.. 7, 23, 35, 111, 119

variability-aware program analysis Directly analyzes the variable code base instead of individually analyzing every possible product variant [Liebig13].. 19–21, 61, 95, 96, 119

Bibliography

[Allen01] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers, 2001.

[Angerer14a] Florian Angerer. *Variability-aware change impact analysis of multi-language product lines*. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 903–906. 2014.

[Angerer14b] Florian Angerer, Herbert Prähofer, Daniela Lettner, Andreas Grimmer, and Paul Grünbacher. *Identifying Inactive Code in Product Lines with Configuration-Aware System Dependence Graphs*. In *Proceedings of the 18th International Software Product Line Conference*, SPLC '14, pages 52–61. 2014.

[Angerer15] Florian Angerer, Andreas Grimmer, Herbert Prähofer, and Paul Grünbacher. *Configuration-Aware Change Impact Analysis*. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pages 385–395. 2015.

[Angerer16] Florian Angerer, Herbert Prähofer, and Paul Grünbacher. *Modular Change Impact Analysis for Configurable Software*. In *Proceedings of the 32th International Conference on Software Maintenance and Evolution*, ICSME '16. 2016.

[Apel09] Sven Apel and Christian Kästner. *An Overview of Feature-Oriented Software Development*. *Journal of Object Technology*, 8(5):49–84, 2009.

[Apel13] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. *Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge*. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, FOSD '13, pages 1–8. 2013.

[Arnold96] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[Ayewah08] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. *Using Static Analysis to Find Bugs*. *IEEE Software*, 25(5):22–29, sep 2008.

[Badri05] Linda Badri, Mourad Badri, and Daniel St-Yves. *Supporting Predictive Change Impact Analysis: A Control Call Graph Based Technique*. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, APSEC '05, pages 167–175. 2005.

[Banker93] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. *Software Complexity and Maintenance Costs*. *Communications of the ACM*, 36(11):81–94, 1993.

[Batory05] Don Batory. *Feature Models, Grammars, and Propositional Formulas*. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC '05, pages 7–20. 2005.

[Berger10] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. *Variability Modeling in the Real: A Perspective from the Operating Systems Domain*. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 73–82. 2010.

[Black01] Sue Black. *Computing ripple effect for software maintenance*. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(4):263–279, 2001.

[Bodden12] Eric Bodden. *Inter-procedural Data-flow Analysis with IFDS/IDE and Soot*. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 3–8. 2012.

[Bodden13] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. *SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years*. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 355–364. 2013.

[Bohner02] Shawn A. Bohner. *Extending Software Change Impact Analysis into COTS Components*. *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.*, 2002.

[Brabrand12] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, and Paulo Borba. *Intraprocedural Dataflow Analysis for Software Product Lines*. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 13–24. 2012.

[Chen01] Kunrong Chen and Václav T. Rajich. *RIPPLES: Tool for Change in Legacy Software*. In *Proceedings of the IEEE International Conference on Software Maintenance. ICSM 2001*, pages 230–239. 2001.

[Clements02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, 2002.

[Cousot77] Patrick Cousot and Radhia Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252. 1977.

[Cousot05] Patrick Cousot. *MIT Course 16.399: Abstract Interpretation*. `http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/`, 2005.

[Czarnecki00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addisson-Wesley, 2000.

[Czarnecki04] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. *Staged Configuration Using Feature Models*. In *Proceedings of the 3rd International Software Product Line Conference*, SPLC '04, pages 266–283. Springer, 2004.

[Czarnecki05a] Krzysztof Czarnecki and Michal Antkiewicz. *Mapping Features to Models: A Template Approach Based on Superimposed Variants*. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, pages 422–437. 2005.

[Czarnecki05b] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. *Staged Configuration through Specialization and Multilevel Configuration of Feature Models*. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[Dhungana11] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. *The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: A Multiple Case Study*. *Automated Software Engineering*, 18(1):77–114, 2011.

[Dubinsky13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. *An Exploratory Study of Cloning in Industrial Software Product Lines*. *2013 17th European Conference on Software Maintenance and Reengineering*, pages 25–34, 2013.

[Egyed10] Alexander Egyed, Florian Graf, and Paul Grünbacher. *Effort and Quality of Recovering Requirements-to-Code Traces: Two Exploratory Experiments*. In *Proceedings of the 18th IEEE International Requirements Engineering Conference*, RE 2010, pages 221–230. 2010.

[Eisenbarth03] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. *Locating Features in Source Code*. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[Ferrante87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. *The Program Dependence Graph and Its Use in Optimization*. *ACM Transactions Programming Languages and Systems*, 9(3):319–349, 1987.

[Fischer14] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. *Enhancing Clone-and-Own with Systematic Reuse for*

Developing Software Variants. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, ICSME '14, pages 391–400. 2014.

[Grimmer16] Andreas Grimmer, Florian Angerer, Herbert Prähofer, and Paul Grünbacher. *Supporting Program Analysis for Non-Mainstream Languages: Experiences and Lessons Learned*. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution and Reengineering*, SANER '16, pages 460–469. 2016.

[Hammer06] Christian Hammer, Jens Krinke, and Gregor Snelting. *Information Flow Control for Java Based on Path Conditions in Dependence Graphs*. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, ISSSE 2006, pages 87–96. 2006.

[Harrold93] Mary J. Harrold, Brian Malloy, and Gregg Rothermel. *Efficient Construction of Program Dependence Graphs*. *ACM SIGSOFT Software Engineering Notes*, 18(3):160–170, 1993.

[Hayes05] Jane Huffman Hayes and Alex Dekhtyar. *Humans in the Traceability Loop: Can't Live With 'Em, Can't Live Without 'Em*. In *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '05, pages 20–23. 2005.

[Heidenreich08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. *FeatureMapper: Mapping Features to Models*. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 943–944. 2008.

[Hofer16] Peter Hofer. *Efficient Execution Time Profiling on the Java Virtual Machine Level*. Phd thesis, Johannes Kepler University Linz, 2016.

[Horwitz90] Susan Horwitz, Thomas Reps, and David Binkley. *Interprocedural Slicing Using Dependence Graphs*. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[Jász08] Judit Jász, Árpád Beszédes, Tibor Gyimóthy, and Václav Rajlich. *Static Execute After/Before as a Replacement of Traditional Software Dependencies*. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '08, pages 137–146. 2008.

[Kästner07] Christian Kästner, Sven Apel, and Don Batory. *A Case Study Implementing Features Using AspectJ*. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 223–232. 2007.

[Kästner08] Christian Kästner, Sven Apel, and Martin Kuhlemann. *Granularity in Software Product Lines*. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 311–320. 2008.

[Kästner09] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don S. Batory. *Guaranteeing Syntactic Correctness for All Product*

Line Variants: A Language-Independent Approach. In *Proceedings of the 47th International Conference on Objects, Models, Components, Patterns*, TOOLS-EUROPE 2009, pages 175–194. 2009.

[Kästner11] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. *Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation*. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824. 2011.

[Kästner12] Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0. it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 54(1):42–46, 2012.

[Kästner14] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. *Variability Mining: Consistent Semi-automatic Detection of Product-Line Features. IEEE Transactions Software Engineering*, 40(1):67–82, 2014.

[Kernighan88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Englewood Cliffs, 1988.

[Kiczales97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-Oriented Programming. ACM Computing Surveys*, 28:220–242, 1997.

[Korel98] Bogdan Korel and Jurgen Rilling. *Program Slicing in Understanding of Large Programs*. In *Proceedings 6th International Workshop on Program Comprehension*, pages 145–152. 1998.

[Korpi07] Jaakko Korpi and Jussi Koskinen. *Supporting Impact Analysis by Program Dependence Graph Based Forward Slicing*. In *Advances and Innovations in Systems, Computing Sciences and Software Engineering*, pages 197–202. Springer Netherlands, 2007.

[Lam11] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. *The Soot framework for Java program analysis: a retrospective*. In *Proceedings of the Cetus Users and Compiler Infastructure Workshop*, CETUS 2011, pages 35–43. 2011.

[Larsen96] Loren Larsen and Mary J. Harrold. *Slicing Object-Oriented Software*. In *Proceedings 18th International Conference on Software Engineering*, pages 495–505. 1996.

[Lehnert11] Steffen Lehnert. *A Taxonomy for Software Change Impact Analysis*. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 41–50. 2011.

[Lettner13] Daniela Lettner, Michael Petruzelka, Rick Rabiser, Florian Angerer, Herbert Prähofer, and Paul Grünbacher. *Custom-developed vs. Model-based Configuration Tools: Experiences from an Industrial Automation Ecosystem*. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC '13 Workshops, pages 52–58. 2013.

[Lettner14a] Daniela Lettner, Florian Angerer, Paul Grünbacher, and Herbert Prähofer. *Software Evolution in an Industrial Automation Ecosystem: An Exploratory Study*. In *Proceedings of the 40th Euromicro Conference Series on Software Engineering and Advanced Applications*, SEAA 2014, pages 336–343. 2014.

[Lettner14b] Daniela Lettner, Florian Angerer, Herbert Prähofer, and Paul Grünbacher. *A Case Study on Software Ecosystem Characteristics in Industrial Automation Software*. In *Proceedings of the International Conference on Software and System Process*, ICSSP 2014, pages 40–49. 2014.

[Liebig10] 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*, ICSE '10, pages 105–114. 2010.

[Liebig13] 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*, ESEC/FSE 2013, pages 81–91. 2013.

[Lillack14] Max Lillack, Christian Kästner, and Eric Bodden. *Tracking Load-time Configuration Options*. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 445–456. 2014.

[Linsbauer13] Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. *Recovering Traceability Between Features and Code in Product Variants*. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 131–140. 2013.

[Linsbauer14] Lukas Linsbauer, Florian Angerer, Paul Grünbacher, Daniela Lettner, Herbert Prähofer, Roberto Lopez-Herrejon, and Alexander Egyed. *Recovering Feature-to-Code Mappings in Mixed-Variability Software Systems*. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, ICSME '14, pages 426–430. 2014.

[Linsbauer16] Lukas Linsbauer. *Managing and Engineering Variability Intensive Systems*. Phd, Johannes Kepler University Linz, 2016.

[Liu06] Jia Liu, Don Batory, and Christian Lengauer. *Feature Oriented Refactoring of Legacy Applications*. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 112–121. 2006.

[Louridas06] Panagiotis Louridas. *Static Code Analysis*. *IEEE Software*, 23(4):58–61, 2006.

[Mayer12] Philip Mayer and Andreas Schröder. *Cross-Language Code Analysis and Refactoring*. In *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, SCAM 2012, pages 94–103. 2012.

[Midtgaard14] Jan Midtgaard, Claus Brabrand, and Andrzej Wąsowski. *Systematic Derivation of Static Analyses for Software Product Lines*. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 181–192. 2014.

[Muchnick97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[Nielson99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, 1999.

[OMG07] OMG. *Abstract syntax tree metamodel (ASTM)*. Technical report, TCS, IBM and others, 2007.

[Ottenstein84] Karl J. Ottenstein and Linda M. Ottenstein. *The program dependence graph in a software development environment*. *ACM SIGPLAN Notices*, 19(5):177–184, 1984.

[Park92] Robert E. Park. *Software Size Measurement: A Framework for Counting Source Statements*. Technical report, Carnegie Mellon University, 1992.

[Payet12] Étienne Payet and Fausto Spoto. *Static analysis of Android programs*. *Information and Software Technology*, 54(11):1192–1201, 2012.

[Petrenko09] Maksym Petrenko and Václav Rajlich. *Variable Granularity for Improving Precision of Impact Analysis*. In *Proceedings of the IEEE 17th International Conference on Program Comprehension*, ICPC '09, pages 10–19. 2009.

[Pohl05] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software Product Line Engineering. Foundations, Principles, and Techniques*. Springer Science & Business Media, 2005.

[Prähofer16] Herbert Prähofer, Florian Angerer, Rudolf Ramler, and Friedrich Grillenberger. *Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application*. *IEEE Transactions on Industrial Informatics*, PP(99):1–10, 2016.

[Rabiser16a] Daniela Rabiser. *Multi-Level Feature Modeling in Industrial Software Ecosystems*. Phd, Johannes Kepler University Linz, 2016.

[Rabiser16b] Daniela Rabiser, Herbert Prähofer, Paul Grünbacher, Michael Petruzelka, Klaus Eder, Florian Angerer, Mario Kromoser, and Andreas Grimmer. *Multi-purpose, multi-level feature modeling of large-scale* *industrial software systems*. *Software {&} Systems Modeling*, pages 1–26, 2016.

[Reisner10] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. *Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems*. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, ICSE '10, pages 445–454. 2010.

[Ren04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. *Chianti: A Tool for Change Impact Analysis of Java Programs*. *ACM SIGPLAN Notices*, 39(10):432–448, 2004.

[Reps95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. *Precise Interprocedural Dataflow Analysis via Graph Reachability*. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61. 1995.

[Rice53] Henry G. Rice. *Classes of Recursively Enumerable Sets and Their Decision Problems*. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[Rubin12] Julia Rubin and Marsha Chechik. *Locating Distinguishing Features Using Diff Sets*. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12, page 242. 2012.

[Rubin13] Julia Rubin and Marsha Chechik. *A Framework for Managing Cloned Product Variants*. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE 2013, pages 1233–1236. 2013.

[Ryder01] Barbara G. Ryder and Frank Tip. *Change Impact Analysis for Object-oriented Programs*. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 46–53. 2001.

[Sagiv96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. *Precise Inerprocedural Dataflow Analysis with Applications to Constant Propagation*. *Theoretical Computer Science*, 167(1–2):131–170, 1996.

[Schaefer10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. *Delta-Oriented Programming of Software Product Lines*. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, pages 77–91. 2010.

[Snelting96] Gregor Snelting. *Combining Slicing and Constraint Solving for Validation of Measurement Software*. In *Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 332–348. Springer Berlin Heidelberg, 1996.

[Sridharan07] Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. *Thin Slicing*. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 112–122. 2007.

[Strein06] Dennis Strein, Hans Kratz, and Welf Löwe. *Cross-Language Program Analysis and Refactoring*. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '06, pages 207–216. 2006.

[Svahnberg05] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. *A Taxonomy of Variability Realization Techniques*. *Software - Practice and Experience*, 35(8):705–754, 2005.

[Tartler09] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-preikschat, and Daniel Lohmann. *Dead or Alive: Finding Zombie Features in the Linux Kernel*. In *Proceedings of the 1st International Workshop on Feature-Oriented Software Development*, FOSD '09, pages 81–86. 2009.

[Thüm14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. *A Classification and Survey of Analysis Strategies for Software Product Lines*. *ACM Computing Surveys (CSUR)*, 47(1):1–45, 2014.

[Tonella03] Paolo Tonella. *Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis*. *IEEE Transactions on Software Engineering*, 29(6):495–509, 2003.

[vonRhein16] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. *Variability Encoding: From Compile-Time to Load-Time Variability*. *Journal of Logical and Algebraic Methods in Programming*, 85(1):125–145, 2016.

[Walkingshaw14] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. *Variational Data Structures: Exploring Tradeoffs in Computing with Variability*. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 213–226. 2014.

[Weiser81] Mark Weiser. *Program Slicing*. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449. 1981.

[Wöß03] Albrecht Wöß, Markus Löberbauer, and Hanspeter Mössenböck. *LL(1) Conflict Resolution in a Recursive Descent Compiler Generator*. In *Proceedings of the Joint Modular Languages Conference*, JMLC 2003, pages 192–201. 2003.

[Xu05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. *A Brief Survey of Program Slicing*. *SIGSOFT Softw. Engineering Notes*, 30(2):1–36, March 2005.

[Xu13] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. *Do Not Blame Users for Misconfigurations*. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 244–259. 2013.

[Xue12] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. *Feature Location in a Collection of Product Variants*. In *Proceedings of the Working Conference on Reverse Engineering*, WCRE '12, pages 145–154. 2012.

[Zhang06] Xiaolan Zhang, Larry Koved, Marco Pistoia, Sam Weber, Trent Jaeger, Guillaume Marceau, and Liangzhao Zeng. *The Case for Analysis Preserving Language Transformation*. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, page 191. 2006.

[Zhang13] Sai Zhang and Michael D. Ernst. *Automated Diagnosis of Software Configuration Errors*. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 312–321. 2013.

[Zheng06] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. *On the Value of Static Analysis for Fault Detection in Software*. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006.

Statutory Declaration