University of Magdeburg

School of Computer Science

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

INF FAKULTÄT FÜR INFORMATIK

Diplomarbeit

A Program Slicing Approach to Feature Identification in Legacy C Code

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Inform. Thomas Thüm
Department of Technical & Business Information Systems

Dr.-Ing. Christian Kästner
Philipps University Marburg,
Department of Computer Science & Mathematics

Author: Steffen Haase, Matr.-Nr.: 174514
Böttgerstr. 49
01129 Dresden
stefhaas@st.ovgu.de

08. Dezember 2011

Contents

List of Figures

List of Tables

Listings

List of Acronyms

AST	Abstract Syntax Tree
CIDE	Colored Integrated Development Environment
FC	Feature Commander
FTP	File Transfer Protocol
IDE	Integrated Development Environment
IRC	Internet Relay Chat
OCaml	Objective Caml

Chapter 1

Introduction

Software engineering aims to increase efficiency in the program development process. Generally methods of modularization within software development are considered to decrease the complexity and, therefore, increase maintainability and reuse of a programs source code. Concepts like separation of concerns describe universal guidelines to be considered in software development to handle complexity. Additionally, frameworks, like domain engineering, were created to provide a general methodology for actively incorporating variability into development efforts. Further, concrete methods, i.e. software product lines, evolved that introduce tools and techniques to incorporate such variability within source code.

While actual development tools and programming languages could not always support the goals and demands of these approaches, efforts were made to adjust and enhance existing technologies accordingly. New programming paradigms, i.e. aspect oriented programming, emerged to encapsulate certain fragments, called concerns, within a program that would otherwise be scattered throughout the code, even though, they theoretically form a single functionality, idea or related structure within the application.

Yet, for developers it is not always possible to account for all potentially relevant software variants. Features or concerns are sometimes added at later times in a manner that contradicts to the initial architecture of the software. These incoherent changes worsen the overall structure of software throughout its evolution [Lehman, 1980]. Therefore, fragments and statements related to a single concern end up scattered throughout the entire source code. This renders source code to be less comprehensible and, thus, harder to maintain or extend [Etzkorn and Davis, 1997]. The identification and classification of related code pieces during code maintenance or later developments are time consuming tasks, which often follow repetitive activities. With more code entangling and no mechanisms to retrieve the true purpose of certain code fragments developers lose traceability [Gotel and Finkelstein, 1994]. This means the connection between conceptual features, i.e., represented by a configuration option, and the actual source code fragments is lost.

The potential of program analysis approaches has been great from the beginning on [Nielson et al., 1999]. Therefore, research efforts created semi-automatic methods to support developers in these tasks [Chen and Rajlich, 2000]. Other approaches increasingly used static and dynamic analysis methods to improve the results of these supporting development tools [Eisenbarth et al., 2003]. Generally, the idea behind these techniques is to automate partially complex tasks for the developers, i.e., tracing program execution or data dependencies.

Other approaches present developers with the possibility of creating different views on existing source code. For instance, visual representation of annotations can increase the comprehension of product-lines up to 43% [Kästner, 2010]. Thus, a reasonable assumption would be that similar visual representations of feature code within existing software would also increase comprehension and, therefore, support development tasks to a certain extent.

The combination of particular static analysis methods and source code presentation techniques, therefore, have great potential to support developers during maintenance tasks. Particularly, slicing algorithms are suited to identify feature related source code parts with a certain accuracy. Additionally, the results can be displayed in a more comprehensible way by using code highlighting techniques. Even simpler analysis algorithms can generate approximations that are similar to what developers achieve in less efficient manual approaches.

Generally, these efforts try to increase development efficiency and code comprehension. Thus, ideal solutions, which are almost impossible to create due to the high complexity, would allow the complete automation of these analytical tasks or even entire code transformations intended to address different software requirements that arise in the future.

1.1 Goals

The major goals of this work is the improvement of program comprehension and automatic code transformation for purposes of transforming dynamic variability into static variability. To achieve this, a subgoal is represented by recovering traceability links between observed configuration options forming a feature and actual source code fragments that implement it.

We have to evaluate how static analysis approaches can be used to support code maintenance tasks and program comprehension. For this, a general process and according steps for the identification of feature code in existing source code needs to be developed. Furthermore, to be able to evaluate the applicability and the necessary amount of work to realize such tools, a prototype needs to be developed as a proof of concept to show that it is possible to automate this process to a certain extent. It comprises the analytical task of the process and it is required to provide an appropriate visual representation of the results. To evaluate the value and potential of this particular approach this prototype is to be compared to manual approaches developers take to execute such an analysis.

1.2 Contributions

In this thesis we contribute a novel process that starts at the manual selection of a configuration option, describes necessary manual steps, executes a static analysis algorithm based on slicing, and, finally, applies the results.

First, we describe the general process, which we divided into four main steps. Understanding these general steps, which are necessary for the identification of feature code, helps to define an environment for the actual analysis step. Within this matter we also determine how dynamic variability is typically represented in existing source code. All this investigation especially leads to the development of the steps that precede the actual analysis process.

Then, we focus on the explanation of the main analysis concept, for which several existing static analysis approaches are combined. We introduce the idea of applying slicing techniques to the task of locating features in existing source code, because it seems to be suitable for this kind of analysis. Additionally, potential enhancements are determined that can be achieved by the application of more precise, but also, more complex techniques. Furthermore, an important focus here is the evaluation of the potential of existing program analysis concepts for this kind of task.

Also, we describe possible scenarios for the application of the analysis results and provide details on how these can be accomplished with the support of existing tools. This helps to determine how these results can improve the development process.

Lastly, to automate the third process step, the analysis algorithm, we design and implement a prototype. This proof of concept allows us to evaluate the concept and demonstrate the entire process on real world projects. The results obtained with this prototype help us to evaluate the two anticipated improvements of development tasks:

Increasing efficiency: The analysis process and, especially, the algorithm are supposed to improve how efficient the developer can be at the investigation of existing code. Even though, the analysis results do not provide a complete set of code fragments to represent entire feature implementing code, the approximation proved to be close to what a manual execution of this task provides to a developer. For this, we present case studies conducted with developers, which show that similar results can be achieved within a fraction of the time needed when it is done manually.

Increased level of program comprehension: By providing analysis information about identified feature code to the developer, we support development tasks concerned with code maintenance or enhancements. Highlighted source code statements provide an additional type of information during usual assessment of existing source code. We show that the analysis results can be displayed comfortably with color highlighted source code.

1.3 Outline

Chapter 2 gives an introduction to basic techniques, concepts and technologies we use for our thesis. These include software product lines, static program analysis, and the code transformation framework, called PUMA.

In Chapter 3 we develop our analysis approach in theory. We describe the general process and provide detailed discussions about how static analysis approaches are applied to the actual analysis algorithm.

The implementation of a prototype on the basis of the theories in the preceding chapter is presented in Chapter 4. Here we put the analysis process into practice and provide scenarios that show how each step can be practically implemented.

This prototype is then evaluated in Chapter 5. Here we compare the results we get to manual investigations. Additionally, we show what improvements of the development process can be expected by applying the implementation to real software projects in the form of case studies.

In Chapter 6 we discuss the problems we encountered and provide possible solutions.

To put this thesis into the context of existing research we cover related work in Chapter 7.

Lastly, Chapter 8 gives a brief conclusion of this thesis.

Chapter 2

Background

This chapter provides fundamental definitions and descriptions of methods, technologies, and concepts used within this thesis. This chapter is, therefore, divided into three major parts, dealing with software development, software analysis, and special technologies used within our work.

Our analysis deals with the identification of source code elements, representing dynamic variability, that are affiliated with certain features or concerns to provide some basic information how these are related to software development in general. To accomplish this, we first describe software product lines and its role within software development (Section 2.1). This section discusses the idea of separation of concerns (Section 2.1.1), which marks a vital concept to generally improve the creation of software. Further, we provide additional insight on that matter by explaining how a certain more problematic kind of concern, called *cross cutting concern* (Section 2.1.2), can appear and not be easily isolated, or separated, respectively. This problem also shares properties with the problem we want to address with our analysis. To finalize this section an overview of possible implementation methods for such software product lines is given (Section 2.1.4). This contributes concrete concepts for possible code transformations.

Secondly, Section 2.2 displays various possibilities to perform static program analysis, which is heavily relied on in the concepts discussed here. While this type of analysis offers a great variety of ideas to focus on several different criteria of source code, it also has to deal with the more or less precise results.

Thirdly, the last section of this chapter, Section 2.3, describes a concrete implementation tool, named PUMA, that we used for the implementation of the prototype. It will discuss structure and capabilities of the framework in a general manner because those also define the preconditions that exist for a possible implementation.

2.1 Software Product Lines

Within the last decades, software products became more diverse and complex, and the fields of application shifted to be more and more intertwined. This evolvement increased the significance of software engineering and the application of its concepts and methods to counter problems that arise in the software development process. Specifically easier software maintenance, higher readability, and the increasing possibilities for reusing parts of the software code are the main criteria concerned within the field.

One of the development concepts to address these issues is called software product lines, which was derived from an existing product line approach typical for the manufacturing industry and applied to the field of software engineering. Software product lines are defined by Northrop [2007] as "a set of software-intensive systems sharing a 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." These software systems, also referred to as *variants*, are similar in certain parts but also comprehend different parts for specific purposes.

The shared resources form a *software platform* [Pohl et al., 2005, p.8,15], which are also called *core assets* [Bass et al., 2003, p.354], uniting all similar *features* of those products. This incorporates the basic idea of *massive soft-*

ware reuse [Van Der Linden et al., 2007] because already created segments of software can be used again for other variants of the same *product line*, yet to be defined.

The idea of reusing software code was not new at the time of bringing this concept to life. Early discussions on handling *program families* were already made by [Parnas, 1976] in the 70s referencing even earlier work done by [Dijkstra, 1976]. Later *domain engineering* was introduced as a framework, which describes fundamental concepts of how domains within software development can be identified and how those can be incorporated in models and, lastly, the implementation. It will be discussed in further detail in Section 2.1.4.

The major difference of a software product line compared to other concepts to increase reusability and decrease complexity is best expressed by the phrases *with reuse* and *for reuse* [Van Der Linden et al., 2007]. While the former can be represented in *small reuse units*, or *ad-hoc reuse*, i.e. at code level [Van Der Linden et al., 2007, p.5] by the use of certain programming paradigms, the latter is a more universal approach involving all aspects of the software development process. *Massive software reuse* and *mass customization* are primary aspects being present throughout planning, designing, and implementing the software systems. A constant concern in this concept is a certain degree of flexibility that needs to be established with the help of existing software requirements and built into the underlying architecture of the software product line [Van Der Linden et al., 2007, p.8].

Just like the product line approach for the manufacturing industry the software product line concept also produces advantages outside of the design scope. Increased code reusability directly decreases the time to market span because a great amount of development and testing has been done already. This increases quality because more evolved software parts are used. It also leads to a decreased amount of required staff, administrative tasks, and other resources, which lastly decreases the cost of a project as well.

2.1.1 Separation of Concerns

Most concepts in software engineering are based on a simple principle referred to as *separation of concerns*. Parnas [1976] and Dijkstra [1976] already discussed this idea of breaking down software into smaller easier managable parts, called *concerns*, to counter problems that arise from the fact that humans cannot handle several concerns very well at the same time. Apel [2007, p.7] describes concerns as a "semantically coherent issue of a problem domain that is of interest". These concerns are elemental for the idea of domain engineering as they need to be identified for the design and later symbolize the very criterias for a *software decomposition*. Apel [2007, p.8] summarizes the benefits for the resulting software being designed with appropriate concerns in mind with better comprehension, easier reuse and maintenance, and more flexible customization.

On this matter Hürsch and Lopes [1995] distinguish between two different levels that separation of concerns is applied on. At *conceptual level* concerns have to be identified and defined as elemental parts that can be clearly distinguished. Whereas, at *implementation level* these separations are manifested in an actual physical organization allowing the isolation of the concerns' code fragments. The former level acts as the conceptual base for the latter, which actually marks the physical entity comprising the benefits mentioned earlier in the form of organized code attributing the appropriate amount of flexibility for the product line, the actual decomposed program.

Technically this separation of concerns can be accomplished in several ways. Programming languages allow for the definition of functions, or classes, again being organized within separate files, packages, or directories. Not always is it possible to account for every identified concern and project it appropriately within the source code. These *cross cutting concerns* [Kiczales et al., 1997] emerge throughout the implementation as "structural relationship[s] between the representations of two concerns" [Apel, 2007, p.21]. The mostly hierachical structures represented within the *modularity* enabled by most modern programming languages are not always compatible with those types of concerns and need to be addressed within new methods.

With this problem in mind Kästner [2010] proposed the *virtual separation of concerns*, a concept ignoring the physical decomposition of certain concerns within the source code. He rather supplies a set of tools combined with an appropriate application of *annotations* to allow on-demand views of concerns (or features) or even entire variants. A somewhat *virtual* separation of concerns is established, while the otherwise confounding usage of *annotations* is accounted for with automatically generated comprehensible representations.

This approach seems to be especially applicable to source code that was written and possibly partially annotated without actual focus on readability or reuse. Less code transformation is necessary later to account for certain concerns and make them visible. These applications correspond strongly to what we consider in this thesis. Because our analysis tries to identify source code entities that belong to a feature or concern, respectively, separation of concerns describes the global concept that is in our consideration.

2.1.2 Crosscutting Concerns

While separation of concerns provides a general approach to structuring source code, it cannot provide for a complete *decomposition.* This means that not all concerns can be accounted for at the time of modularization because during implementation several become intertwined. Kiczales et al. [1997] refers to these as *crosscutting concerns* because the source code blocks of one basic software functionality is simply *cross cut* by code artifacts of another concern.

The main reason for the existence of such structural breaches lies in the very nature of certain programming paradigms, or programming languages, respectively. The specific kind of abstraction that is done upon a particular piece of software must be compatible with the sort of abstraction mechanisms offered by a particular programming language in order to allow a complete decomposition. In example, while a requirement can be decomposed by features, the decomposition at code level can be accomplished by *objects* (in object oriented paradigm) or functions (in procedural paradigm) [Tarr et al., 1999].

Additionally, these traditional programming paradigms bear a common limitation, manifested in the hierarchical structured abstraction that only allows decomposition in one *dominant dimension.* As a result code representations of a concern contradicting a specific type of decomposition cannot comply with the intended structure completely. It becomes *scattered* throughout implementations of other concerns and *tangled* in the form of modules consisting of several representations.

Possible solutions to address crosscutting concerns were provided by Kiczales et al. [1997] in the form of *aspect oriented programming* (AOP), which describes a paradigm allowing the actual separation of otherwise tangled concern representations. Additionally, Kästner [2010] proposes methods and tools that rely on mere annotations within the source code and provide separated views of programmatically intertwined concerns to the programmer.

Crosscutting concerns need to be considered in this thesis because our analysis also discovered dynamic variabilities, whose representation in the source code were scattered throughout several modules. Therefore, descriptions given here can be used in the attempt to explain these observations.

2.1.3 Domain Engineering

To be able to create a construct like a software product line, a specified process is needed that comprises all necessary steps. Usual software development technologies can be improved by the adoption of reuse [Arango and

Prieto-Diaz, 1991, p.9]. Reusable software systems and fragments need to be developed with certain structures in mind to allow for quick and easy customization and the demanded reuse for possible future applications within the same domain. Commonly shared and well defined *features* need to be the result of identified and structured requirements, which exist for the software family that needs to be developed. To determine these, the developers start with an analysis of the domain, expressing the results within a domain model, which again will be used to draft and implement the actual software product line.

[Czarnecki, 1998, p.44] identifies two relevant elements existing within domain engineering, the *problem space* and the *solution space*. The former symbolizes all "valid system specifications" in a domain, whereas the latter is manifested within the actual software systems or implementation concepts. A major intention for Domain Engineering is to support the mapping between these two spaces, and preferably automatize it to a certain extent. This is not a trivial process due to a natural discrepancy in objectives set for a system specification and a concrete software system. While the former aims toward a direct description of the entirety of a problem, the latter reflects the general effort of development to create rather granular components that allow for as many combinations as possible, and, therefore, a higher degree of flexibility.

Just like software can be classified by the area of application, i.e. accounting systems, production control systems, monitoring systems, etc., also software parts can be categorized, i.e. by their purpose or role within the software system, for instance database systems, graphical user interface libraries, etc. [Czarnecki, 1998, p.33]. This classification builds the basic idea of creating a domain with its requirements (problem space), and creating the actual software implementations to perform certain functions (solution space).

A categorization is possible as long as several software systems or parts share a common set of characteristics. Czarnecki [1998] discusses these "areas organized around classes of systems or parts of systems" and refers to them as *domains*. These domains mark the space in which a certain software family exists with all required knowledge, data, and information that can potentially be helpful to create a certain degree of reusability. Not only similarities or commonalities are important within the domain, as they are reflected within the earlier mentioned core assets. Also the differences need to be a major concern because they are the focus for future *variation points* [Bass et al., 2003, p.360].

Within this particular field of software engineering, an appropriate definition of the term *domain* is given by Kang [1990, p.2]:

> "Domain: A set of current and future applications, which share a set of common capabilities and data."

It respects the consideration of future applications, which can be considered one of the main concerns of Domain Engineering. The software systems existing in one domain will be required to fit the needs of several more or less different application fields, or many different customers, respectively. To achieve these goals in an ordered manner, Czarnecki [1998, p.33] breaks Domain Engineering into three major *phases*, *Domain Analysis, Domain Design, and Domain Implementation.*

Domain Analysis identifies, gathers, and analyzes knowledge specific for a certain domain. It marks a systematic approach to reliably capture information that is needed presently and may be needed in the future [Arango and Prieto-Diaz, 1991, p.10] and structure it appropriately. The latter is mostly accomplished in the form of a *domain model*, which represents relevant "*common* and *variable* properties of the systems" [Czarnecki, 1998, p.35]. The variable properties are usually described in the form of a *feature model* where all reusable and configurable elements (*features*) and their relations or dependencies, respectively, are documented.

In the next phase, *domain design*, these domain and feature models are used to develop a general architecture for the software that is to be implemented. This architecture builds a guideline for the implementation and gives an overview to the developers. Therefore, it might contain several types of views to adequately represent necessary elements and their relationships in appropriate form [Czarnecki, 1998, p.38]. Buschmann et al. [2007] discussed such an architecture in the context of software product lines, trivially called product-line architecture. He defined it as a "software architecture that serves as a common basis for a whole family of related software products". It creates a stable foundation comprising the commonalities reaching across all elements and the set of all configurable features that represent the resulting variability. One specific configuration, an instance of this architecture, consists of a selected set of those features and describes one possibles variant in the final implementation [Pohl et al., 2005, p.317].

The architecture needs to be a high-level representation of the software product line, addressing all relevant requirements but keeping all descriptions as far away from an actual implementation as possible. This ensures a flexible structure to be established from the top [Czarnecki, 1998, p.39]. Flexibility, here, has two aspects to it. On one hand, the architecture needs to allow for a certain degree of flexible implementation because certain requirements can only be met at that level, i.e. performance requirements. Secondly, the flexibility of a software family must also be reflected, therefore, the representation of variability must be explicitly visible.

The final phase, called *domain implementation*, transforms the architecture into a real instance of a software product line. Components are implemented and tested, and the actual variability needs to be realized within this software system. This allows for a quick and flexible *application engineering* in the future where actual customer tailored software is derived from the product line. The ideal scenario would allow a direct mapping of future customer requirements to those that were identified for the domain during the domain

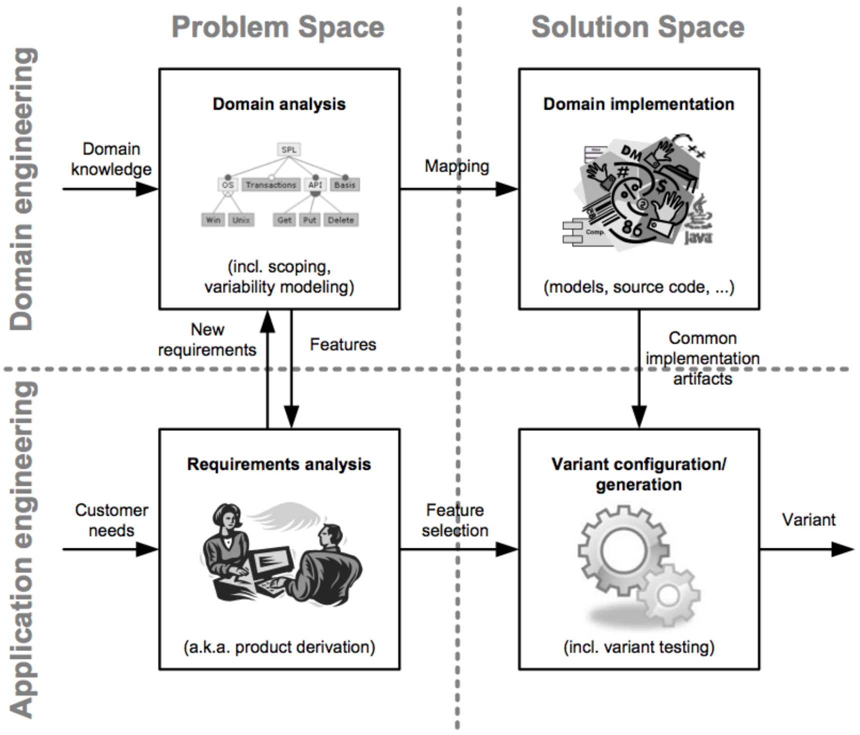


Figure 2.1: An idealized overview of the interaction of domain engineering and application engineering processes. [Kästner, 2010, p.9].

analysis (problem space) because they are very likely to be already present in the product line implementation (solution space) [Kästner, 2010, p.9].

Figure 2.1 shows the framework that is formed by these concepts. The interactions depicted in this overview resemble the ideal scenario mentioned earlier. Particular approaches and tools for each phase can be applied.

2.1.4 Implementation of Software Product Lines

The decision of implementing a software product line within a company must be made individually and the benefit needs to be evaluated appropriately. Aspects that can be considered as beneficial because of the usage of a software product line are very diverse and can be found not just within the entire development process, but also inside organizational properties.

An example for the former are the *requirements*, which could have already been covered in preceding work for possible variants in the domain analysis phase. Even if this was only partially the case, extensive time for analysis can be saved. This is similar for the *modeling* phase and, furthermore, the resulting *architecture* that has already been established for a similar software system. Logically, also simple software reuse benefits can be yield due to the fact that *components*, which exist already, can simply be reused. Even if a certain degree of customization, for instance by adjusting parameters, needs to take place, concrete implementations in the form of data structures and algorithms can be left untouched. This also includes time for *testing* that can be saved directly because tests for certain components have already been done, or indirectly because tests have already been prepared. This means test cases have been established before, test processes are defined, and test results as a basis for comparison do already exist.

On the organizational side experience has been gained, too. *Processes* are familiar, results from earlier *plannings* exist in the form of schedules or budgets, and as a direct outcome of all these prior aspects less workforce and time is necessary to create other variants.

To be able to actually utilize the benefits to the greatest extent possible the implementation of an actual software product line differentiates itself from usual application implementation in the fact that variability needs to be represented within those software systems in an appropriate manner. The usual coding of components is accompanied by the concern of representing identified features from the designed architecture within the final product line. Therefore, developers have several mechanisms to be utilized for implementing such flexibility.

Variability

Variability describes the "ability to change or customize a system" [Babar et al., 2010]. Variability in software, therefore, refers to any possibility within a software system or a set of several software systems that allows for a certain degree of customization. Variants within a software product line, for example, are created or, rather, generated, by the utilization of specific *variability mechanisms* [Northrop, 2007, p.87]. These mechanisms are distinguished by the specific type of variability, which directly defines another property, the time, when the variation takes place. Different works within the field discuss several possible definitions of these types. A certain set of those were gathered by Jacobson et al. [1997] and include mechanisms that can commonly be found in many resources. Table 2.1 shows these mechanisms and also comprises the corresponding property "Time of Specialization" that is typical to the individual mechanism type.

This list is similar to the list that Svahnberg and Bosch [2000] established specifically for architecture-based support for variability in product lines. It does not contain the "Uses" and "Template Instantiation" mechanisms but adds yet another commonly used type, which they call "Compile-time selection of different implementations" [Northrop, 2007, p.69]. This type is essential to this thesis and will be described in further detail later. There are many ways of implementing compile time variability, the most common being the utilization of certain preprocessor directives, such as `#ifdef`, within the programming language `C` and its derivative `C++`. There are also similar solutions that were developed for other programming languages such as Java.

A need for a certain variability can be identified at any time during the development phase. Just like trivial *variation points* can be identified right from the requirements, for example during the domain analysis phase, more covert ones can arise during the domain definition, or the development phases [Bass et al., 2003, p.360]. This does not just refer to the initial implementation but also in later developments that might be necessary for certain variants. Wi-

Mechanism	Time of Specialization	Type of Variability
Inheritance	At class definition time	Specialization is done by modifying or adding to existing definitions. Example: LongDistanceCall inherits from PhoneCall.
Extension	At requirements time	One use of a system can be defined by adding to the definition of another use. Example: WithdrawalTransaction extends BasicTransaction.
Uses	At requirements time	One use of a system can be defined by including the functionality of another use. Example: WithdrawalTransaction uses the Authentication use.
Configuration	Previous to runtime	A separate resource, such as file, is used to specialize the component. Example: JavaBeans properties file
Parameters	At component implementation time	A functional definition is written in terms of unbound elements that are supplied when actual use is made of the definition. Example: calculatePriority(Rule)
Template instantiation	At component implementation time	A type specification is written in terms of unbound elements that are supplied when actual use is made of the specification. Example: ExceptionHandler<Container>
Generation	Before or during runtime	A tool produces definitions from user input. Example: Configuration wizard

Table 2.1: Variability mechanisms. [Northrop, 2007, p.88]

jnstra [2000] describes another mechanism that they utilize in their medical imaging systems.

The *service component frameworks* create a scaffolding for customizable software products. These frameworks are categorized into two different types. *White-box* frameworks share similarities with the earlier mentioned inheritance mechanism because they utilize the concept of dynamic binding, which means concrete information about the framework's internals are necessary for the development.

On the other hand, *black-box frameworks* mostly put well defined interfaces to practice. Frameworks' internals are irrelevant for developers because all functionalities are held within components that exclusively use these interfaces to offer their functionality. The product, which obviously itself can be considered to be a component within an even larger project, is a composition of one or more frameworks and several plug-ins that are connected into them.

2.2 Static Program Analysis

While domain engineering provides methods and conventions for systematic software development, it mostly affects or improves, respectively, the architectural levels of this process. On the lower levels, concerning concrete implementations and coding details, *static program analysis* provides tools and frameworks that allow to check the actual behavior of implemented portions of the application.

Static analysis, also called compile time analysis, refers to techniques that predict a program's behavior before its execution takes place [Nielson et al., 1999]. In contrast to *dynamic analysis*, where run-time information is used to investigate the behavior of a program, this form of analysis relies on compile-time information only, such as the program code.

Typical information that is gathered in this process is *data flow*, which identifies the relationships between data objects and how values are passed among them. This is also related to the investigation of *control flow*, which ideally represents all possible orders in which statements of the program are executed (2.2.2). Thirdly, *abstract interpretation* can be used to approximate a program's behavior through *partial evaluation* and the creation of abstract formalization systems (2.2.2). Lastly, an approach called *slicing*, which comprises several of the other analysis methods, is used to calculate a connected portion of the program that affects values at a specific point of interest (2.2.3).

Application of static program analysis can be found in compiler optimization, i.e., to reduce redundant computations in the resulting program or eliminate unnecessary operations. It is also used for program validation, where the actual behavior of a program's implementation is compared to the intensions that were defined priorly to avoid malicious or simply unintended actions during execution.

The analysis described in this thesis heavily relies on concepts that are part of static program analysis. Thus, the definitions and explanations provided here give the reader an overview of the relevant techniques that exist in the field.

2.2.1 Approximation

The ideal goal of program analysis lies in computing behavior for *all* possible inputs and code structures. This is not possible because static analysis takes place before the actual input is known. In the field of computer science this problem is known as Rice's Theorem, which states "all but trivial properties about a program are undecidable." [Andersen, 1994]

To work around the restrictions of undecidable properties within a program analysis approaches need to generate *approximations*. In the area of investigating control flow of a program this could lead to the application of assumptions, for example the acceptance of the premise that all source code statements are actually reachable. While this assumption makes processing source code easier, it also has the drawback that cases where a fragment of the code is actually not reachable are also considered during the analysis. This results in a less accurate representation of the control flow. Of course this claim is not valid for *all* programs but for *some*, which shows that the accuracy of the results vary [Andersen, 1994].

Approximation will play a role in our analysis algorithm implementation. Due to time and resource restrictions, simpler algorithms will be used, which are more likely to generate *conservative* results.

2.2.2 Control Flow and Data Flow Analysis

During program development interesting questions concerning the program behaviour can arise. A subset of these questions are concerned with the flow that is generated during execution of the program [Mossin, 1997]. Flow, here, refers to the order in which source code entities gain control and, thus, are processed during run-time. While certain analysis approaches in this area are concerned with the control flow of expressions or statements (*control flow analysis*), others are, i.e., rather focused on the implication it has on other source code entities, like data items (*data flow analysis*) or functions (*function call analysis*).

A usual depiction and formulation of these relationships are *flow graphs* [Nielson et al., 1999]. Each node represents an *elemental block* and is connected to other nodes by edges, which represent the passing of control. Elemental blocks can, of course, be any of the code objects that are to be investigated during the analysis. Thus, in a control flow graph, a path within this graph represents one possible order in which statements can be executed, while in data flow analysis a path rather shows the dependencies between data objects.

For this thesis especially control and data flow approaches were used to define and implement our analysis algorithm. With focus on configuration variables data flow analysis supports the process of determining data dependencies throughout a program. Additionally, control flow analysis allows us to identify what statements are influenced how and, thus, need to be marked accordingly.

Control Flow

Because each function can directly be identified by its lexical identifier, control flow can be exactly determined in programming languages that have no *higher-order functions* [Midtgaard, 2007]. These are functions that either take another function as an input or return a function as a result. In more advanced imperative, functional, or object-oriented programming languages this information is not directly visible because the actual function call cannot directly be determined from the application text or source code, respectively [Midtgaard, 2007, Nielson et al., 1999].

Data Flow

Especially relationships between those items and statements that alter or rely on them are of value in both directions. On one hand it is desirable to determine which uses of a variable were influenced by a certain definition. On the other hand, it can also be of value to identify data definitions that provide values to a particular data use [Allen and Cocke, 1976]. To further clarify these relationships, *data definitions* refer to statements that modify a data entity, such as a variable. Accordingly, *data uses* are expressions that employ a data item without actually altering it.

Approaches

Theoretical formulations of control and data flow analysis were created in several different ways. Many similar of these formulations exist because they are modifications or combinations of each other. An overview of control flow formulations is given by Midtgaard [2007], which summarizes several research efforts. The major groups are mentioned and explained here briefly.

The first group considers *constraint based* approaches. These usually consist of two phases. Within the first phase, constraints are determined that solutions to the analysis need to satisfy. In the second phase, these solutions are actually calculated. Depending on the type of constraints, different concrete formulations were developed, i.e., *grammar based analysis*, which uses set-constraints [Heintze and Jaffar, 1990], or *closure analysis*, in which *equation constraints* are extracted and the resulting equation systems are solved [Henglein, 1992, Nielson et al., 1999]. Later, Cousot and Cousot [1995] showed that these constraint-based approaches mentioned are merely instances of *abstract interpretation*.

The second group is called *type-based flow analysis*. It additionally uses the inherent information of types that lies in typed higher-order programs [Midtgaard, 2007] to further approximate the flow of information. A standard application scenario of type-based flow analysis is type inference, which allows the automatic deduction of types for data items at compile time.

Thirdly, a formulation can be done in the form of *abstract interpretation*. Cousot and Cousot [1977] introduced this formal analysis methodology, which deducts behavioral information from a system of formal semantics. Generally, this approach creates an abstract representation of program operations by abstracting original objects to a meta level and performing transformations to create a computable formal analysis system. Thus, the results gained from an abstract execution of such a created system represent approximate results of actual computations during the execution of original programs.

A closely related area is described by the name *partial evaluation*, where programs are executed by an interpreter with only a part of the input data provided. Thus, the program code is partially evaluated. Certain variables in the source code, then, can be resolved to particular values, which leaves a *residual program* as a result [Jones et al., 1993]. Partial evaluation can also be formulated within the context of abstract interpretation Hermenegildo [1999].

Differentiation

Midtgaard [2007] compares these properties by introducing a measure of *sensitivity* that helps to categorize approaches according to the precision they can achieve. While the first property is mentioned for completeness, the second categorization is more common and will be used predominantly later to classify the approaches.

One distinction is based on the fact that an analysis accurately follows the control flow of a program's source code, in which case it is referred to as *flow-sensitive*. *Flow-insensitive* concepts, on the other hand, roughly approximate the control flow by defining assumptions in advance.

A more prevalent distinction separates *context-sensitive* from *context-insensitive* analyses. While the former takes contextual information at the time of calls into account, the latter does not. Specifically whithin the field of control flow analysis, these properties are also referred to as *polyvariant* and *monovariant* Midtgaard [2007].

2.2.3 Slicing

The goal of our analysis is to compute such a slice with respect to configuration variables that are related to a particular feature or concern.

Program slicing is a method for abstracting from program's source code, that was introduced by Weiser [1981]. It allows the reduction of a program, ideally to a minimal extent, with the consideration of a subset of the original behavior. Thus, a generated *slice* represents an independent program that holds a specified behavior of the original application.

This desired behavior is usually reduced to a set of values, that are computed at a certain point of interest [Tip, 1995]. Thus, to provide a particular behavior to any kind of slicing algorithm, relevant information needs to be selected in advance, which is summarized under the term *slicing criterion.* For specific programming languages these usually consist of a set of variables and the location of the point of interest within the source code. The goal of a slicing algorithm, then, is, to calculate a subset of statements that consists of all necessary portions of the program that either affect this slicing criterion or are affected by it.

Finding minimal slices is generally not possible, which is due to the same reasons that were discussed in static analysis (2.2.1). Because all slicing algorithms rely on concepts and methods from static analysis, like data flow and control flow analysis, the resulting approaches can only be as precise as the methods are that they are based on.

2.2.4 Properties of Slicing Algorithms

Within the field of slicing analysis the distinction between two types of slicing was established. These are *static slicing*, which merely relies on compile-time information for the computation of a slice, and *dynamic slicing*, which is done with the consideration of run-time data. For the latter approach, of course, a set of predefined input values is necessary to actually execute the code, which limits the analysis to a certain degree. Harman and Hierons [2001] also mentions a less common approach, called *conditioned slicing*, which represents a combined approach between dynamic and static slicing.

Another property aims to distinguish between the directions that dependencies within the source code are investigated. In a *forward slice* all code blocks and statements remain that are somehow affected by the variable in the slicing criterion. Vice versa, in a *backward slice* all statements are kept that have an impact on a particular variable [Tip, 1995].

Lastly, a slice can be characterized by the type of transformation that can take place. To generate a *syntax-observing slice*, an algorithm can only remove parts of the source code that have been found to not affect the point of interest, at all. On the other hand, an algorithm can be created that is allowed to perform any kind of syntactic transformation, which preserves all desired semantic constraints. This approach is named *amorphous slicing*.

2.3 The **PUMA** Parser

The PUMA framework is a major part within our implementation of the prototype. This section will give a brief introduction to the library itself, its capabilities, and its role in the analysis process.

PUMA is a code transformation framework, provided as a library, that consists of a parser, a set of data structures that handle file and project management, and several mechanisms that allow various types of code transformation. The framework itself was developed in C++ and is freely available as part of the AspectC++ Project, which is still maintained and enhanced [Urban et al., 2010].

The framework is capable of managing entire source code structures by handling them as projects. These projects consist of directories holding the actual source code files. Within this structure these files are parsed and the resulting *translation units*, which are a tokenized representation of a single source code file, is kept underneath the file data structures within those projects. Even though files are the main resource for source code, there are also other supported possibilities typical for UNIX systems, i.e. *Pipe commands.*

The essential feature within the PUMA framework is its parsing capabilities. Supporting the languages C, C++, and various dialects, it provides lexical and syntactical checks before creating the actually *abstract syntax tree* (AST). To offer a more complete parsing of these languages, a C preprocessor was also integrated. It supports common substitutions, source code inclusions, and complex macros.

Of course, methods are supplied to execute common semantic C and C++ checks upon the created ASTs. On top, these methods produce an optional semantic database, which can be helpful for further analysis.

Beyond these parsing capabilities PUMA provides extensive transformation tools. These allow for simple token manipulations or even intensive complex transformation of entire ASTs. To further support these actions special checking mechanisms are included that identify possible transformation conflicts.

Urban [2002] gives a great overview of the PUMA framework by identifying six levels of operation:

The PUMA framework provides a reliable C and C++ parser with a wide range of extension possibilities. The source code of the framework and example implementations for code analysis and transformation are available. Especially the semantic database or class database, respectively, provides helpful tools for our analysis. Additionally, code transformation is supported

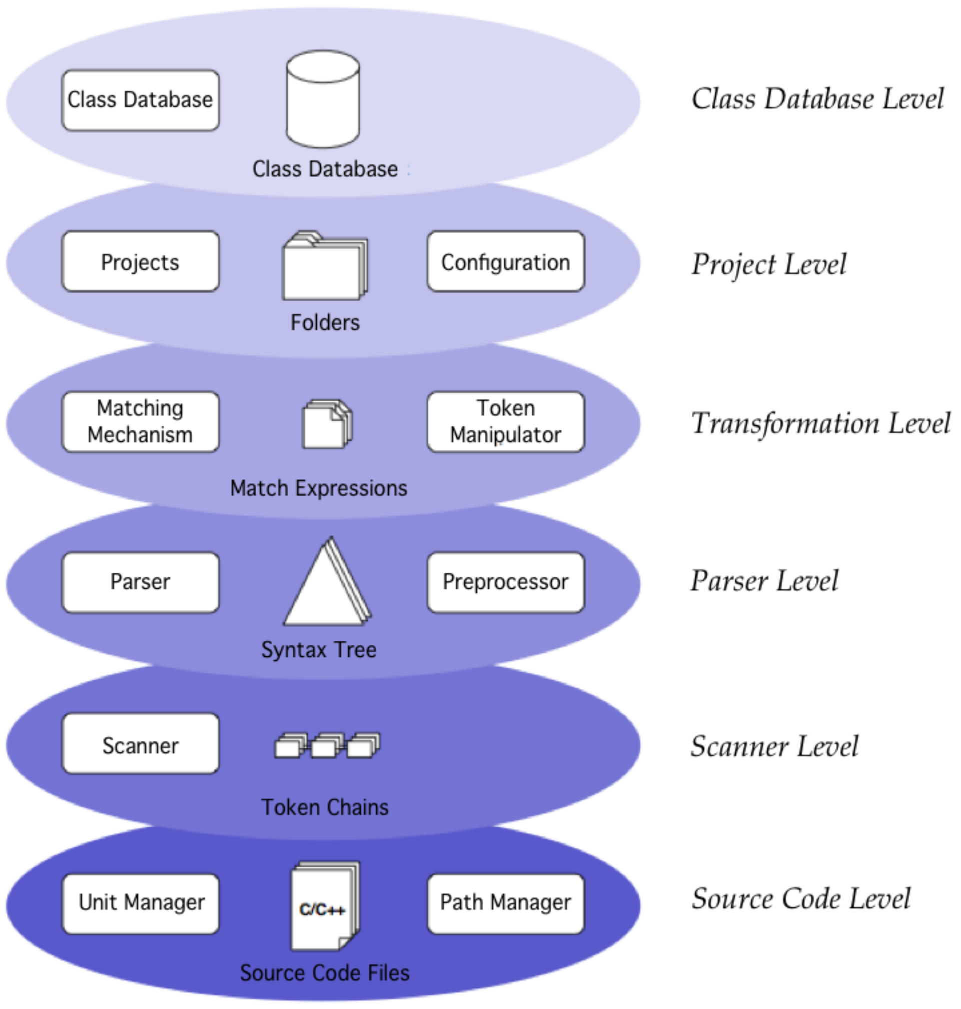


Figure 2.2: Operation levels of the PUMA framework [Urban, 2002]

extensively, which is a further goal we have in mind and will be practically experimented with in the future. Lastly, the potential for C++ parsing is maintained, even though, this initial work focuses mainly on ANSI C.

Figure 2.2 illustrates the hierarchical levels on which the PUMA framework was developed. This architecture allows quick modifications and extensions to be developed to cope with several structures during application use, like files, statements, items from the semantic database, preprocessor information, or transformation functions.

Chapter 3

Concept

The idea in this project is to use a simple conservative slicing algorithm to identify concerns or features within an existing software system that were implemented as dynamic variability. The main focus for this analysis, therefore, lies on the identification of code branches and significant code dependencies that exist and, thus, represent these features. After this process, these interdependent code parts can be either displayed or even automatically transformed, i.e., to convert dynamic variability into a static one. To test and evaluate the theoretical constructs, we designed and implemented a prototypical analysis tool (see chapter 4). As proof of concept and for evaluation, we will focus on essential phases discussed in this chapter, solely, because of the complexity of more complete approaches.

The general analysis process is separated into several phases, where each requires a certain input and generates a certain output, that again might represent an input for the next step. This thesis will not explore all of these in detail simply because some of the phases can be considered as special research fields of their own. Additionally, not all of them are considered to be automated, in fact, for certain steps an explanation will be given to discuss the usefulness, applicability, and limitation of automation.

Generally, first a single code entity needs to be identified, which implements some kind of control over a particular variability. Then, dependent code parts are being computed from that, which directly represent either a partial or even the complete implementation of that particular feature. Figure 3.1 depicts the individual phases necessary for the analysis.

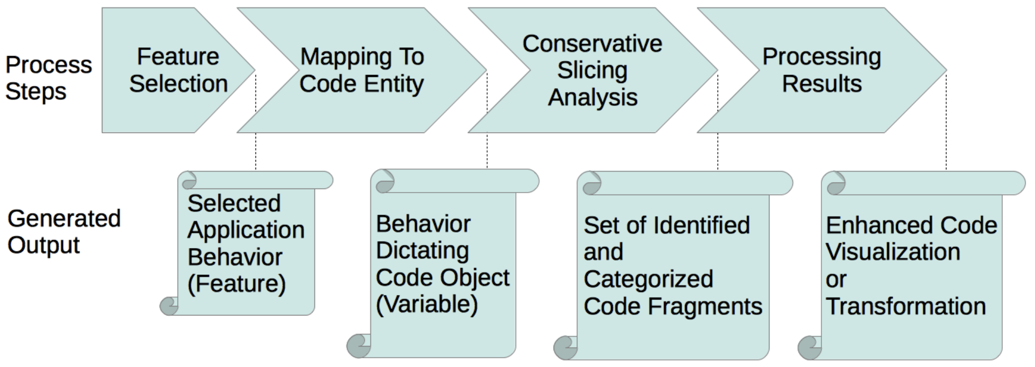

Process Steps
Feature Selection
Mapping To Code Entity
Conservative Slicing Analysis
Processing Results
Generated Output
Selected Application Behavior (Feature)
Behavior Dictating Code Object (Variable)
Set of Identified and Categorized Code Fragments
Enhanced Code Visualization or Transformation

Figure 3.1: The main analysis phases and their generated output.

Selection of an existing feature: The feature that is going to be identified within the code needs to be selected by a user or a domain expert. This feature is represented by an application behavior that can be observed during execution. An example could be an optional encryption functionality in a server application.

Mapping to a behavior dictating entity: A developer needs to identify the according code entity that controls the observed behavior of the program leading to the execution of source code belonging to the feature that was determined in the first phase. Ideally, this leads to a single variable holding certain values, which are used to control the application's behavior. This could be a boolean variable within a server application that holds either the value `true` during execution when it is enabled by the user or the value `false` when it is disabled.

Slicing analysis: In this step all traceability links are identified that connect the selected option to the according variability in the source code. For this, the priorly determined source code entity, i.e. a variable, is used to determine dependencies to other source code fragments. These relationships form the entire feature implementation. Further, statements of this code subset are categorized into three states that are supplied for further investigation in later steps. These are *belonging to a feature*, *not belonging to a feature*, and *possibly belonging to a feature.* Here an `if`-statement relying on a previously selected boolean variable would be considered a part of a feature, because the code block underneath is only executed, if the variable holds the value `true`. Also, a statement defining this variable would belong to the feature.

Processing the results: All the code fragments that were identified and marked by the algorithm can now be used for further processing. Generally, there are at least two possibilities. Logically, the results can simply be displayed in an appropriate manner, which could be used to support developers, i.e., by marking interdependent code statements. With more complicated and complex efforts, they can also be used for a code transformation algorithm, for instance to allow a (semi-)automated conversion of dynamic variability into a static variability.

As an example we look at the grep project and its configuration parameters because this project offers several configuration options, which are not implemented in a centralized manner, but rather scattered throughout the entire source code. This command line tool takes a text as input from the standard input and filters it to generate output that matches a certain pattern. A normal pattern can be a single phrase that is passed as a parameter. Grep would display all lines of the input text that have the provided word in them. We will use this command line tool as a continuous example throughout the phases that are explained in further detail within the next sections.

3.1 Selection of an Existing Feature

Because the analysis in this thesis aims toward locating relevant source code statements that implement a certain feature, some kind of identification needs to take place in advance. This task marks a mere selection of a special behavior within the software, which can simply be based on observations being made during the execution of the software itself. This is consistent with the definition of a feature, which describes it as a part of a piece of software that is directly visible to a user (2.1).

Generally, any functionality of a software can be part of the selection, but our experience showed that especially configurable parts of applications tend to be particularly suitable for this kind of analysis. This is due to the fact that a trivial method often used to implement a certain degree of configurability, or variability, respectively, is to employ simple variables that hold special values at run-time to control the actual behavior according to the configuration.

Obvious examples for this kind of configuration are command line parameters that are offered to specify or modify the execution of programs intentionally. This sort of flexibility resembles one kind of dynamic variability, which was discussed in section 2.1.4. Especially primitive configuration possibilities that simply switch functionalities *on* or *off* proved to be ideal for this analysis, as their representation within the source code is mostly restricted to two possible values, just like they are typically held by Boolean variables. Section 3.3.3 further explains why this is helpful to reduce the complexity of the later analysis.

Command line tools, as widely spread within Linux distributions, offer a great variety of options in the form of mandatory or optional command line parameters or values set in configuration files. These often follow generally prevalent formatting conventions. In many maintained software projects these parameters, however passed to the application during execution, are well explained in the associated documentations. Thus, a logical possibility to identify and select existing features within a software is to scan the documentation coming with the software for any kinds of configuration possible.

For the grep example a specific behavior, that can be configured by enabling or disabling it, is case sensitivity. By default grep does consider case sensitivity, thus, the feature is turned off and the case of the pattern is not ignored for the filtering process. The user can enable it by providing the parameter "`-i`" in the command line. This selection will be used in the next step.

Grep offers more configuration parameters like "`-c`", which counts the number of matches, "`-T`", which aligns displayed tabs, or "`-H`", which displays corresponding file names.

3.2 Mapping to Source Code Entity

A vital step in the discussed analysis process is the identification of essential source code entities that specifically implement a configuration option, and thus, dynamic variability. Mostly, this step has to be a manual one. The selection of a certain feature within an application, which was done in the last phase, has to somehow be linked to parts of the software's source code. This is necessary in order to actually communicate this selection to any potential analysis tools in later phases. As already mentioned, for that purpose we would like to use some kind of slicing algorithm, which generally requires the definition of a *slicing criteria* as a starting point.

Conditional statements influence the control flow by the evaluation of expressions, which are mostly comprised of variables. Mostly, because there are other artifacts that can provide a value, as well, i.e. functions or number representations directly. These usually rely on variables themselves to actually hold the values, but resolving those dependencies is more complex and will be discussed later (see Section 3.3.1). In contrast, variable dependencies can be observed directly. The mere presence of a variable name in an expression of a conditional statement creates a dependency of any code branch created by this statement on that particular variable. The exact nature of this relationship, though, requires further analytical processes.

This phase seems similar to a usual debugging approach a developer would go through to locate essential code elements responsible for the software malfunction to be analyzed. Source code fragments of interest are singled out by following code traces that seem significant to a certain application behavior. The most interesting code parts, therefore, are *conditional statements*, which directly alter the *control flow* of the program based on some condition.

Since it seems logical for a software developer to use variables to handle and store configuration values during the execution of the program, it is also reasonable to use those directly in such conditional statements. Thus, whenever a significant branch of source code statements has been found, an analysis of the conditions under which they are executed would be necessary. Ideally, these conditions can be resolved to a single variable or a set of interdependent variables, which not only makes the analysis itself easier but the implications for further steps, as well. Usually naming conventions that developers follow to increase maintainability and readability of their source codes give hints on what option certain variables might be related to. So, any kind of development or code documentation would be an appropriate initial source of valuable information.

Lastly, the source code itself needs to be focused on. As mentioned, particularly variables seem to be a valid tool for storing and providing values for later reference - later in the sense of a future point in time of the program execution.

Throughout our observations we found that a typical shape for representing configuration in source code are global variables, which store configuration information or parameters provided by a user. Very often these values are parsed from whatever source they come from, i.e. the command line, and assigned to variables during early stages of the execution. These values usually do not change during the execution, and because they are quite often needed at several different parts of the code, providing them globally is a preferred choice of implementation.

The case sensitivity parameter in `Grep` is represented in the source code. Following the execution starting at the ”`main.c`” file in the `main()` function the determination of passed parameters is done with the help of a certain function, called `get_nondigit_option()`. The code displayed in example 3.1 shows how the parameters are extracted by this function and then evaluated by the `switch`-statement on Line 3. The previously selected feature that was symbolized by the "`-i`" is directly represented by the constant on Line 6. As a result of providing the `i`-parameter to the application, the value 1, which corresponds to the value `true`, is assigned to the global variable `match_icase` on Line 8, which obviously represents this particular configuration option in the source code. This feature variable will be focused on in the next analysis phase and, thus, be provided as the slicing criterion.

Listing 3.1: Mapping parameter to according code object (`Grep`).

```
1  prepend_default_options (getenv ("GREP_OPTIONS"), &argc, &argv);
2  [...]
3  while ((opt = get_nondigit_option (argc, argv, &default_context)) != -1)
4      switch (opt)
5      [...]
6      case 'i':
7      case 'y':  /* For old-timers . . . */
8          match_icase = 1;
9          break;
10 [...]
```

Automation of The Process Step

The first two phases of the analysis address the need of certain predefined criteria that the analysis relies on. Considering automation of the selection process seems to be impossible because a selection is not a static behavior but rather relies on the context and the individual choice of a developer. On the contrary, the mapping of options to code entities seems to be a less arbitrary task.

We found an interesting approach, that addresses the particular variability of program configuration options and describes a static extraction process to automatically identify it. In their paper Rabkin and Katz [2011] describe a combination of methods, including call-graph analysis, string matching approaches, and other heuristics created from observations they made in real open source projects.

The approach itself is based on identifying a key-value style configuration and the underlying model of implementation by matching it to programming patterns that were observed priorly. In a second step the identified options are then categorized upon their type and provided to the user. The categorization also address if options are actually used throughout the software or are merely leftovers from previous software versions.

This analysis is additionally backed with a case study providing results drawn from a prototype implementation of this approach allowing the analysis of Java projects. They were able to find 95% of the original program configuration options.

While this implementation was done to analyze Java source code only, the implication for our work is that a certain degree of automation is possible. The partial identification of program configuration options could provide an additional source of selection for program variability. On the other hand, the mentioned approach extensively supports the identification of specific source code entities, like variables, which could replace an otherwise manual process. Designing and implementing this automation for C projects is outside the scope of this thesis.

3.3 Slicing Analysis

The third phase marks the actual analysis that uses the previously identified source code objects to calculate an actual conservative slice of the source code, ideally, containing all instructions of the code that are somehow dependent on this source object. It is the most complex and most resource intensive part of the process.

The decision to apply a slicing algorithm for the identification of features within an existing source code seems quite trivial. A feature is merely implemented by adding instructions to a source code that describe the specific intended behavior. These instructions "interact" by processing input and generating output for each other, therefore, implicit dependencies arise.

Whenever a feature is included within a software, all instructions and code fragments belonging to this feature need to be executed in the intended order. If the starting point of this particular feature is executed, all necessary instructions implementing it need to be present for execution, as well. Vice versa, if a feature becomes obsolete and is either removed or simply switched off, the initial instruction of this feature is never executed making all dependent code fragments irrelevant because they cannot be executed, either.

An important note on these initial instructions is that they can be directly related to the objects identified in the previous step. In example, let **feat_A** be a variable, which has been determined to be an essential variable for the execution of a certain feature. In this case every single conditional instruction that includes this variable in its condition can be seen as an initial instruction. The execution of a specific feature code merely depends on the value that is assigned to this variable at execution time and the condition itself.

Example 3.2 shows a single variable conditional statement and the two control branches it is creating. **statement_list_A** is only executed if the value stored in variable **feat_A** renders the expression to be *true*. In this case the branch symbolized by **statement_list_B** is not executed. Both branches directly rely on variable **feat_A** and so a direct dependency is formed. Whenever the initial statement represented by the first line is executed, either of the compound statements will be executed, as well. Vice versa, no execution of either of the branches will take place without an execution of the initial `if`-statement.

Additionally, it is obvious that both, **statement_list_A** and **statement_list_C**, belong to the same feature. Their execution relies on the same condition, thus, both of the `if`-statements can be considered as initial instructions representing entry points to code blocks related to the particular feature.

Listing 3.2: Initial instruction in the form of a single variable and its code branch.

```
if(feat_A) {
        [statement_list_A]
}
else {
        [statement_list_B]
}
[...]
if(feat_A) {
        [statement_list_C]
}
```

These restrictions and relationships are vital for further implications for the analysis. On one hand, it shows what additional analysis steps might need consideration, or more precise, how control flow needs to be determined in each case. On the other hand, it states what meaning the results of this analysis have.

For example 3.2 the dependencies described earlier imply that the code in **statement_list_A** belongs to **feat_A**. Further, **statement_list_B** cannot be part of the same concern, in fact, it is only relevant whenever **feat_A** is not. For this block more contextual information is necessary to do a proper categorization. The existing form of dependency suggests it is representing a clear alternative to whatever behavior **feat_A** describes. Suppose, as a result of this analysis, **feat_A** was identified and is to be discarded from the code. The question arises how this would also affect **statement_list_B** and its necessity to remain in the code. These aspects particularly are discussed in further detail in Section 3.4.2.

3.3.1 Classification

Because different types of dependencies arise during the analysis, we decided to not only mark but categorize the identified code fragments. Example 3.2 in the last section already showed two types of dependency in a simple case. A certain code block can be directly linked to a feature, while another piece of code obviously must not be part of it. This distinction arises from the `if-else`-construct in the programming language. Whatever condition is used within the `if`-statement will apply to the corresponding compound statement. Implicitly, to the compound statement belonging to the `else`-statement the negation of the condition applies. Thus, actually the condition or rather the type of condition, is relevant to the subsequent categorization.

In this consideration two main types of conditions can be differentiated, because they have dissimilar implications on the categorization of the underlying code fragments. These types are *single variable* and *multi-variable conditions*. As their names suggest, the distinction is based upon the count of variables present in a conditional expression. While a categorization in one case can be made easily, the other does require additional attention.

Clear and Unclear Single Variable Conditions

The analytically simpler type comprises a single variable in the condition. Thus, the control flow merely relies on a single variable or the underlying value, respectively. The maximum amount of possible branches at this control statement depends on the data type stored within the relevant variable. This variable can either carry a **boolean**, which would limit the variability at this point to two options, i.e. enabling or disabling a certain feature. Or, the variable carries another data type provided by the programming language, which would allow for variability with more than two options but is limited by the size of the actual data type itself. While the former condition type would simply consist of the variable and the optional unary operation **not** (”!”), the latter requires a binary operation , i.e. **equals** (”==”), and another comparison operand representing a particular option or a set of those.

Listing 3.3: Beispielcode

```
1 if(feat_A) {...}
2
3 or
4
5 if(feat_A>5) {...}
```

The classification here is not always clear. Both statements show a single variable condition, though, the different type of usage suggests a different exercise for implementing variability. The `if`-statement on Line 1 simply relies on the boolean value in variable `feat_A`, which makes the affiliation to `feature_A` obvious. In this case the condition merely consists of a single variable, whose value is resolved to either `true` or `false`. Therefore, in the example we can categorize the underlying code block to *belonging to the feature.* In a case where the condition also comprises the `not` operator, i.e. in the expression ”`!feat_A`”, we categorize the code block as *not belonging to the feature.*

The expression on Line 5 is a little more complicated. Using a configuration variable in combination with a comparison operator offers the developer the possibility to select between several options at that point of the code. It also opens up the possibility to unite code blocks that are used by several features. This is displayed in the following scenario.

Suppose a developer wants to create a program with a feature that has several variations. These can be configured during execution and, therefore, were implemented in a combined manner, because they have certain code fragments in common. The comparison operator in the control statement's condition can be used to create exactly that. A possible application for this kind of dynamic variability would be the implementation of a logging feature, which can be configured to operate at several different levels, i.e. displaying information to a user directly and writing it to a file.

Example 3.4 shows such a scenario with a feature variable `logging`. Depending on the value that it holds, different combinations of the defined code branches can be executed during run-time. For the value 1 the branch defined on Line 1 and 5 are executed. For the value 2, Lines 3 and 5 define the behavior during run-time.

Listing 3.4: Single variable in comparison conditions.

```
1 if(logging==1) {...} //logging into a file
2 [...]
3 if(logging==2) {...} //logging by displaying to the user
4 [...]
5 if(logging>=1) {...} //logging with both possibilities at the same time
```

It does not always make sense to group these feature variations together. Therefore, we decided to also classify these code blocks as *unclear* or *possibly belonging to the feature*, respectively. This classification also seems suitable considering that the intentions of the developer are not clear at the time of the analysis. Using a comparison operator can also be used to implement a simple enabling/disabling mechanism, which would clearly be equal to a boolean variable. Without further investigation this cannot be determined.

Unclear Multi-Variable Conditions

Multi-variable conditions comprise several different variables and, therefore, more complex logical expressions. For the analysis to be completed, these expressions require more resources and more analytical methods to extract the implications for the code underneath each conditional strain. Ideally, the expressions only consist of variables representing feature configurations because then, the analysis of the condition can implicitly represent the relationship of these features.

Example 3.5 displays a possible select statement, that uses multiple variables in the condition. This condition seems to represent parts of the implementation of a feature A, that do not apply if another feature B is enabled. A more detailed analysis of the occurrences of such conditions throughout a source code would suggest a direct dependency between feature A and feature B, where either feature can only be enabled if the other is disabled.

Listing 3.5: Conditions with complex condition expression.

```
if(feat_A && !feat_B) {...}
```

While these conditions are not the major focus in this thesis possible solutions using partial evaluation through abstract interpretation or symbolic evaluation to enhance the results and increase approximation (see 2.2.2).

We assign corresponding code parts with this property directly to the category *possibly belonging to feature*, because at this point we cannot make certain claims without further investigation. This classification allows the code fragments to be marked in a way that in future steps they can be reconsidered manually by a developer.

3.3.2 The Relevant Control Mechanisms in C

Consider a subset of instructions within an application's source code that is never reached unless a feature controlling variable holds a certain specific value. Therefore, this subset of code lines can only be part of the specific feature implementation, because it is only executed whenever a specific run-time configuration exists. A run-time configuration in this case consists of a set of variables, which hold a specific value at a certain time. For our analysis this would mean, each source code object, i.e. a variable or a method, that is only reached during execution whenever certain conditions are being met can be considered a part of a specific feature implementation.

The main focus for this simple kind of analysis, therefore, lies on source code statements that create a branch within the execution tree in dependency of a particular value stored in a relevant variable. In this thesis we only consider the programming languages ANSI C because the control flow mechanisms are less complex. Because C++ extends the programming language C by an object oriented paradigm, the results of the investigation in this thesis also apply to C++. Yet, to gain more accurate results for the same analysis, this approach would have to be extended as well to account for the control mechanisms that are additionally provided by the object oriented paradigm.

The decision to look at this language particularly is related to the number of existing analysis, parsing, and transformation tools, and especially the great variety of available applications and their source code. The open-source community, which greatly implements in C and C++, also provides a good amount of developer documentation, that is very helpful within all phases of this analysis process. Lastly, a lot of research that we found to be related to this kind of work has been done with focus on C and C++, as well.

The programming language C provides the developer with a small set of control structures that allow for the creation of code branches or dynamic variability, respectively. These *control statements* influence the order in which the statements of the source code are executed [Businger, 1988]. Table 3.1 shows all relevant control statements for this analysis.

The *selective statements* are particularly interesting because they allow for a selective behavior of an application relying on specific conditions, which are directly visible. The *iteration statements*, on the other hand, seem less valuable to this analysis because they tend to be less appropriate for selection

Type	Statement	Description

selective	if
	else
	else-if
		A conditional statement that is followed by a boolean expression and a statement sequence, which is also called *compound statement*. The execution of the compound statements merely relies on the evaluation of the *boolean expressions* at run-time. `If`-statements alone allow for controlling the execution a single compound statement. Combinations with `else` statements or `else-if`-statements, respectively, allow the creation of alternative branches of code.

	if expression	Conditional statements, similar to `if`-statements, consisting of expressions, which directly return a value. These expressions allow two results that can be returned depending on if the boolean expression, delimited by a *ternary operator*, usually represented by the symbol '?', renders to be `true`, or `false`.
	switch/case	A conditional statement that compares a given value (i.e. from a variable) to a set of specified constants. Each constant or sets of them have statement sequences assigned to them.
iterative	while/do	A code construct that allows the repetition of a code sequence until a boolean expression renders to be `false`.
	for	A construct allowing code blocks to be repeatedly executed for a specified number of iterations. The number of repetitions can also result from arbitrary expressions and statement.

Table 3.1: List of control statements in `C`. [Businger, 1988]

purposes by their nature. Nonetheless, they can be used in such way and, therefore, must be considered.

For similar reasons this thesis' focus lies on the identification of conditional statements that evaluate single relevant variables. This restriction not just makes it possible to reduce the cost of doing the analysis but also makes use of the inert structural similarity that exists between the general convention of implementing such variability and this analysis that tries to identify it.

Selective Statements

Certain implications of selective statements were already partially explained in the previous subsections. Generally, we categorize the statements and underlying compound statements by the properties the according conditions show.

`If`-statements that have a single variable condition can be directly associated with the particular feature corresponding to the relevant variable. If the unary operator NOT, "!", was found, the relationship of the categorization is simply negated. Same applies to compound statements underneath the appropriate `Else`-statement. In example 3.6 and 3.7 these scenarios are illustrated. `Statement_list_A` will be categorized *as being part of* ***`feature_A`***, while the `statement_list_B` will be explicitly considered as *not being part of that feature.*

Listing 3.6: Single variable `if-else`-statement.

```
if(feat_A) {
        [statement_list_A]
}
else {
        [statement_list_B]
}
```

Vice versa, in example 3.7 `statement_list_A` will be categorized as *not being part of* ***`feature_A`***, while the `statement_list_B` is properly considered to *be part of that feature.*

Listing 3.7: Negated single variable condition.

```
if(!feat_B) {
        [statement_list_A]
}
else {
        [statement_list_B]
}
```

Because a direct association becomes more complicated in complex multi-variable conditions, we applied the third more vague category symbolizing the state of *uncertainty*. It allows us to capture the uncertain affiliation and possibly provide it to additional analysis steps at a later time.

Besides `if`-statements, we also found the application of `switch`-statements to be a common way to create variability. This is specifically applied when several alternatives needed to be implemented, which generates more than two possible code branches. Because there is no naive way of telling apart certain feature implementations, we categorize the entire `switch`-statement as being unclear. This generally makes sense because the switch statement semantically offers the possibility to create variations of control flow in dependency of one variable. Therefore, several implementations can be distinguished, which all relate to a single feature. On the contrary, an interpretation is possible where only a single case actually represents the implementation of a feature. This case is not covered in our analysis.

This more accurate representation, though, which accounts for each of the variations individually, would need some kind of evaluation. This can either happen on a higher level by comparing the actual names of the constant variables in the `case`-statements or by evaluating the actual values, which is a rather wasteful approach due to the static nature of the constants. The latter option would only account for the rare case in which a mix of constant variables and their direct values is used in several `switch`-constructs.

Example 3.8 depicts an example of such a `switch-case`-structure. Our analysis would mark the entire `switch`-statement and categorize it as having a possible feature_A affiliation. More precise analyses could identify the relationships in detail, i.e., that functionY() on Line 5 would belong to both, variation_X and variation_Y. Additionally, a possible alternative interpretation is shown, where the variation_X case is handled individually in the `if`-statement. This raises the question if feature_A is actually the feature in focus or if each variation represents its own feature.

Listing 3.8: Example for handling `switch`-statements.

```
1  switch(feature_A)
2  {
3          case variation_X:
4          case variation_Y:
5                  functionY();
6                  break;
7          case variation_Z:
8                  functionZ();
9                  break;
10 }
11
12 if(feature_A==variation_X)
13 {
14         functionX();
15 }
```

Iterative Statements

The implementation of variability can also be accomplished by the use of iterative statements. These are usually used to implement repetition but can imitate the behavior of `if`-statements, as well, because source code within the compound of such iteration statements can be executed when the iteration occurs once or more, or not at all, in cases where value and condition combinations would result in an iteration count of zero. This kind of flow control seems less probable because implementing a desired dynamic variability in such way would render the source code to be less comprehensible. This seems not to be a general approach a developer would chose but should be considered, nonetheless.

In the investigated examples we have not found examples of such an application. It is, though, possible to create scenarios where this might become a logical solution. Imagine the implementation of a keep alive functionality for network connections. Usually, these repeatedly send messages through the network to sustain a priorly established connection to another peer, that otherwise would get terminated after a specified time within an idle state. Similarly, the implementation of a network retry functionality can make use of the capabilities a `for`-statement provides. The configuration of the particular application allows the specification of retry attempts on a failed connection. Is it set to the value 0 the retry feature itself is disabled, is it set to any positive value, it is enabled and provides the configurable value for the amount of retry attempts to be made at the same time.

Example 3.9 shows these scenarios where the keep-alive functionality is controlled by the timed_feature variable. As long as this feature is enabled during execution the function is executed repeatedly. The feature variable retry_feature, here, not just enables or disables the particular feature but also provides another configuration value that defines the number of repetitions that will take place during execution.

Listing 3.9: Possible scenarios for use of feature variables within iterative statements.

```
while(timed_feature)
{
        sleep(500);
        send_keepalive_message();
}

for(int i=0;retry_feature>i;i++)
{
        retry_connection();
}
```

In both cases we identify the use of variables in focus within the conditions. We mark the entire code block including the control statement and the according compound statements. Additionally, we mark those as possible candidates. A more accurate analysis, though, would suggest a similar treatment as we propose during the investigation of `if`-statements and their conditions.

3.3.3 Determining Control Flow

In order to extract the necessary dependencies between statements it is necessary to determine in which order they are executed and where relevant key data structures have an impact on the execution. This is done by the application of control flow and data flow analysis. Tip [1995] describes a variety of slicing algorithms based upon several different approaches. Generally all of those aggregate information in some kind of dependency graph and then more or less accurately compute a certain slice dependent on a selected slicing criterion.

While he covers both, forward and backward slicing, our analysis only applies the former concept. This decision seems logical, because this algorithm actually extracts a statement subset of a program that directly or indirectly relies on a specific variable. Even more restricting is our focus on the control influence this particular variable has upon certain statements regulating the mere execution of those. The problem we try to address, therefore, directly calls for a forward-slicing algorithm.

In his paper Tip [1995] also discusses a major concern several times, the completeness of a calculated slice. None of the concepts are capable of calculating a minimal precise subset of statements. Each algorithm described has a certain limitation and degree of approximation and, therefore, calculates a somewhat inaccurate slice.

While many suggested algorithms rely on the generation of dependency trees, information flow graphs, or data flow graphs, our concept does not create such trees directly. To decrease complexity of the analysis and simplify the implementation of a prototype we use a simple recursive algorithm. In order to still gather enough information to generate a slice it needs to traverse the source codes AST, find relevant variables within conditional statements, and mark all branches belonging to those statements accordingly.

The path that the algorithm takes to walk the AST follows a naive static *control flow* analysis approach. Beginning at one statement which is considered to be the starting point of the application, i.e. a function named "`main`", the algorithm follows the trivial code execution path defined by code blocks, conditional statements, and function calls. For this thesis we will only consider the mentioned entities, even though, there are other more subtle mechanics that also cause branching in the control flow, such as function pointers. This makes the approach more conservative, which results in less code being visited throughout the analysis. Visiting the AST in this manner allows us to determine possible paths that the execution of the application could follow and, thus, approximately symbolize the order in which the statements can occur.

Because we have certain restrictions, i.e. a single variable to focus on, not all possible paths have to be considered. Code branches, which clearly cannot be influenced by a particular variable, do not need to be followed during the tracing of the execution.

Function Calls

For function calls we identified two types, that can exist in two scenarios. Each combination of type and scenario implies a different behavior for the analysis to continue. The first type is represented by calls that do not pass a relevant variable as a parameter. The algorithm does not need to further investigate the code in the particular function definition because none of the statements within the definition lie in the scope of the relevant variable, which makes a dependency impossible.

The second type of function calls passes a relevant variable and, therefore, needs to be further examined. This is necessary because the relevance of the variable is passed on to one of the arguments defined for the function. Because of the expansion of the scope, it is possible that control statements in the definition rely on the particular variable, now embodied in a local variable within the function body.

Example 3.10 illustrates several function calls, where each has a different relationship to the relevant variable `feat_A`. The definition of `function1()` on Line 3 does not need to be visited for dependencies on this variable because these statements do not lie within the scope of the variable. On the contrary, because the feature variable is passed on Line 4 the definition of `function2()` needs to be visited with the local variable in focus that the relevant variable was passed to. This will identify the conditional statement on Line 12 to be part of the feature implementation. Additionally, implications arising from this kind of variable use will be discussed in in the next paragraph. Lastly, the function calls on Line 7 and 8 are occurring within a code branch whose execution completely relies on the value in `feat_A`. Besides applying the analysis rules that were just explained another concern arises, which will be discussed now.

Listing 3.10: Relevance of other variables

```
int feat_A,b;

function1(b);
function2(feat_A);
[...]
if(feat_A) {
        function3(b);
        function4(feat_A);
}

void function2(int isEnabled) {
        [...]
        if(isEnabled)
        {
                [...];
        }
}
```

As mentioned earlier, function calls can occur either outside of or within a compound block of a control statement that relies on the feature variable. While the former does not create a necessity for further investigation, the latter forms a special case. Independent of the particular call type, the dependency here lies in the mere usage of the function.

Implications. An example for this is displayed in figure 3.11. Executing our algorithm on this example will identify the conditional statement at Line 5 and its compound statement spanning from Line 6 to 10. Further, let `function_A` be a part of the implementation of `feature_A`. This implies that no function call outside of the scope of the specific feature occurs. For the purpose of this example let the call on Line 8 be the only call of `function_A()`

within the entire project. At the time of analysis, our algorithm does not determine that the execution of `function_A()` directly relies on the execution of the compound statement underneath the corresponding conditional statement. The marked code fragments would not include the definition and declaration of `function_A()`, as they should for a more complete result. If this feature was to be removed from the source code on the basis of identified statements, these remaining code parts would not interfere with the rest of the source code. It does, though, create an overhead by leaving unnecessary artifacts bloating the resulting source code. This clearly represents a typical result of an approximation.

Listing 3.11: Relevance of other variables

```
int main () {
        int feat_A;
        feat_A=...;
        [...]
        if(feat_A)
        {
                [...]
                function_A(...);
                [...]
        }
}
void function_A(...) {
        [...]
}
```

Solution Luckily, this specific type of problem can be accounted for by gathering function calls within identified code blocks during the traversal of the AST. Succeeding the analysis, a search for other calls of this function throughout the entire project would reveal the actual dependency. If no other occurrences of according calls are found outside of the scope of similar compound statements, the function can be considered part of the feature and, thus, be categorized appropriately. Because this check needs to be done after the analysis, it can be considered an additional step to further enhance the results from this analysis. Existing development environments, support tools, and compilers are able to identify unused code fragments, already.

This algorithm represents a naive incremental approach. While most slicing algorithms use dependency graphs or control flow graphs, we generate this information while visiting all code entities. This allows us to make ad-hoc decisions and focus on a smaller amount of dependencies. In graph based approaches the entire dependency graph needs to be created, which is a resource intensive task. While this is a very accurate representation, it is not suitable for the simpler restricted analysis we pursue because the complexity of such graphs would be unnecessary when focus lies on simpler boolean configuration variables and the features they control. Also, the incremental character makes this approach more scalable and, thus, easier to extent. Very simple scenarios can be covered with less effort while complex scenarios call for more complex analytical algorithms.

Handling Loops

Another consideration in the area of function call tracing needs to be the possible occurrence of loops. Because the analysis actually follows the control flow and function calls to a certain extend, a mechanism is needed to ensure the analysis does not go on for ever. In our analysis this is accomplished by tracking the visits the algorithm does for certain function definition. This is done on a variable basis. Was a function visited because a relevant variable was passed to it, a flag denotes that in an appropriate data structure. This ensures the function is not visited again with the same variable in focus.

Generally, this would be needed for each parameter individually, which can be passed to that particular function. This is due to the fact that each parameter gets assigned to a different local variable in the function definition's scope. Thus, different selection statements would be affected and must be considered.

Example 3.12 represents such a scenario. The configuration variable feat_A is passed to the function in two different ways on Lines 4 and 5. The first call would make the selective statement on Line 8 and its compound statement possible feature implementation parts.

The second call on Line 5 would also result in the function definition being visited by the analysis algorithm. This would render the selective statement on Line 11 to also be part of the feature implementation. Additionally, the algorithm would continue the analysis for variable y on Line 12 as it, again, represents a function call with a relevant variable, that is passed.

Listing 3.12: Loop scenario with

```
1  int main () {
2          int feat_A;
3
4          function(feat_A,0);
5          function(0,feat_A);
6  }
7  void function(int x, int y) {
8          if(x) {
9                  [statement_list] }
10
11         if(y)
12                 function(x,y);
13 }
```

We do not include this individual inspection in our analysis because it simply seems unlikely that a relevant configuration variable is passed to a function through several different parameters.

Data Flow

Additionally, it is necessary to trace where values of relevant variables might travel. We found that many times they are either assigned to other variables, or even passed on to functions in the form of parameters. This was also the case for globally provided variables, probably as a result of different developers working on a single project or because priorly implemented methods are being used without modification. Nevertheless, the analysis does require some sort of *data flow analysis* to address this phenomenon. Fundamental details on such examinations were described by Allen and Cocke [1976] and usually based on prior control flow analysis.

Tracking the flow of certain variable values can be challenging. A value can be passed on to another variable, which then becomes relevant to the analysis, as well. Therefore, the analysis starts by focusing on a single variable but might end up having to address several. This already bears complications. The relevance of this new variable is not existing in all code fragments but merely in the once executed after it has been assigned a significant value. Example 3.13 shows how the relevance of a variable might change within a code block. The variable we look at for the analysis is `feat_A`. While the conditional statement on Line 3 is not relevant, the same type of statement becomes important in Line 7. The relevance, therefore, was transported from `feat_A` onto `b` at the assign statement on Line 6. Continued analysis must account for variable `b` from that point on.

Listing 3.13: Relevance of other variables

```
int feat_A, int b;
[...]
if(b) {
        [...]
}
b=feat_A;
if(b) {
        [...]
}
```

Because we use an incremental approach this problem is addressed automatically. Data flow is implicitly determined with control flow by the algorithm. The more accurate the control flow analysis is, the more accurate becomes the analysis of the data flow.

3.3.4 Storing the Results

Throughout the analysis the algorithm gathers a list of statements and statement blocks for each feature and each code file. This list also holds the categorization information for these statements or statement blocks. Because this information is gathered incrementally, the particular form of storing the results can be adjusted to a desired processing method. Therefore, the result of a single analysis represents a granular categorization of statements within a source code project.

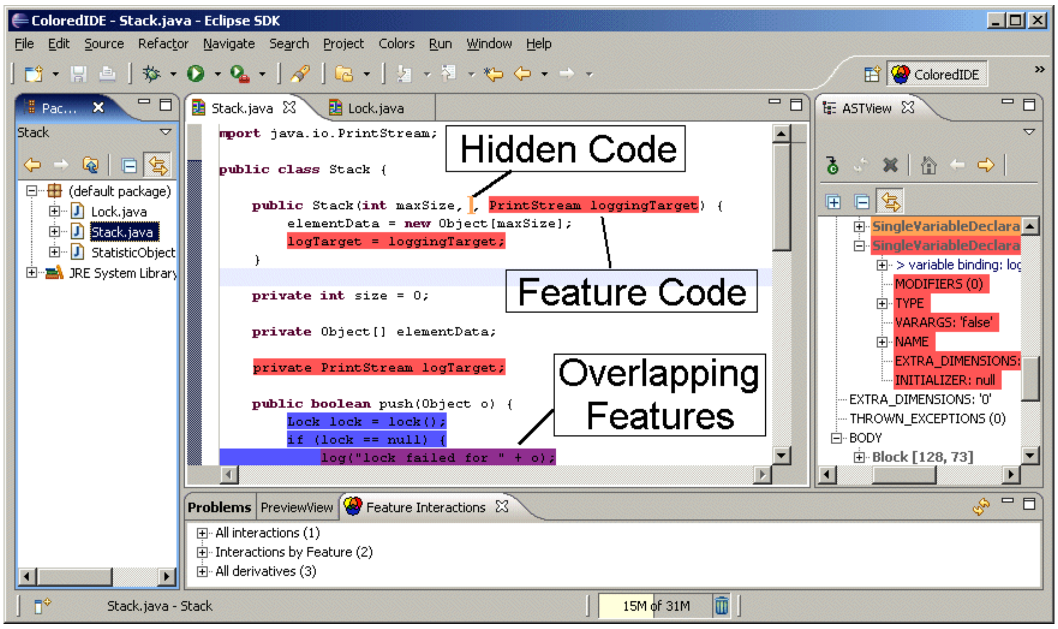
3.4 Processing the Results

After the slicing analysis identified interconnected code branches and categorized these in a particular way, these results need to be used in a valuable manner. There are several ways to make the collected and calculated data accessible to a developer. Generally, these can be categorized in two major application scenarios. Either a visual representation is created, where results are prepared and structured appropriately to support a developer during development, maintenance, or simply for code understanding. Or, the identified structures are used as input for other mechanisms which more or less automatically transform the source code in a desired way and, therefore, become an active part in the development process to increase efficiency.

3.4.1 Visual Representation

The identification and representation of concerns in source code can be a valuable information for application developers. Aspect oriented programming, for example, tries to provide developers with a method to thoroughly apply separation of concerns when other paradigms lack the capabilities to do so. Other approaches suggest tools and mechanics to restructure annotated source code to create more comprehensible views of otherwise scattered code fragments [Favre, 1996, Kästner, 2010]. Separation of concerns itself represents an abstract form of categorizing source code parts according to their contextual roles. Therefore, the value of such supporting analytical development tools is undisputed.

The kind of information that was gathered in the previous step has a similar purpose and, thus, can be treated the same way. Code blocks have been categorized according to their dependencies on particular code entities, i.e. variables. A logical step, therefore, is to make these categories visible to the user. This can be accomplished by coloring code blocks accordingly. Working on a single concern is made easier because code fragments that are contextually connected are visually represented in the same way. Kästner et al. [2008] describe the use of background colors in the Colored Integrated Development Environment (CIDE) to visualize feature affiliation of source code lines. An example of this is shown in figure 3.2. For this purpose also a code visualization tool, named Feature Commander (FC), can be used, which is



```
mport java.io.PrintStream;

public class Stack {

    public Stack(int maxSize, |, PrintStream loggingTarget) {
        elementData = new Object[maxSize];
        logTarget = loggingTarget;
    }

    private int size = 0;

    private Object[] elementData;

    private PrintStream logTarget;

    public boolean push(Object o) {
        Lock lock = lock();
        if (lock == null) {
            log("lock failed for " + o);
```

Figure 3.2: The CIDE withen the Eclipse IDE. Kaestner [2010]

shown in detail within section 4.3 along with figure 4.2 depicting the result visualization in the FC tool of a grep analysis.

Integrated into a development environment, i.e. Eclipse[1], this kind of analysis can enhance existing support tools, like highlighting references for certain variables. In the current stage Eclipse simply marks statements that contain the use of a certain preselected variable. While this kind of support only presents direct dependencies of statements on a certain code entity, our analysis could provide additional dependencies to the user. The highlighting can be extended to entire code blocks, whose execution relies on a single variable, or even scattered code fragments that were identified to show a certain degree of dependency. The potential value of introducing program dependency graphs into development environments was already discussed by Ottenstein and Ottenstein [1984]. The advantages would be relevant in editing, translation, debugging, and for the application of program metrics.

[1] `http://www.eclipse.org`

The analysis maps source code parts to a variety of categories (3.3.1). Since it is very complicated or even impossible to decide on what implication a certain code structure has on the mapping, we end up with several categories that are related to a single concern. These subordinate separations should be addressed in the visual representation as well. For instance, when several concerns are displayed it seems convenient to chose similar colors for subcategories that are believed to belong to a single concern. Another option would be to merge these subdivisions automatically by the application of certain rules or manually by a developer. Experienced developers, ideally knowledgeable of the existing code, could then complete the categorization. With the uncertainties eliminated, the completed categorization could prove to be a valid and more precise input for a possible source transformation.

3.4.2 Source Code Transformation

Another possibility to use the results of the feature-identification analysis is to transform the according source code with the help of certain rules. While, of course, it is possible to manually modify identified fragments with a certain intent, a more interesting idea is an automatic approach. A simple intention of a modification would be the *removal of an entire feature.* This can be accomplished by simply taking the results from the previous phase and delete lines accordingly, so they wont be present at compilation time.

This residual version of the program source code would in fact represent a result that can be obtained from a partial evaluator 2.2.2. This specialized version of the program is created by assigning a constant static input value for the program, i.e. for a certain variable. This renders parts of the code that were previously dynamic to be static. Thus, these parts of the code can be transformed statically, which is called *partial evaluation* of a computation process. The transformation mentioned in the last paragraph would be accomplished by assigning the value `false` or 0, respectively, to a feature enabling variable. All affected code fragments can be transformed accordingly.

On the other hand, enabling the feature would also create a residual version by assigning the value `true` or a value greater than 1, respectively, to the according variable.

Another more flexible way is to use annotations, i.e. preprocessor directives, to switch code blocks on or off for compilation. This represents an approach to turn the dynamic, or run-time, variability, into a static, or compile time, variability. Features that were dynamically enabled or disabled during the execution of a program on the basis of variable values are now turned to features that are enabled or disabled during compilation. Resulting programs, then, either do not contain any of the implementation statements for a feature or always include them, thus, always carrying the behavior that was existent in the original program with the configuration for a feature enabled. For this purpose research within the field of refactoring [Fowler and Beck, 1999] is valuable, because it can provide transformation frameworks and rules that leave original behavior intact. Particularly refactoring annotations [Tansey and Tilevich, 2008] seems to be a suitable application because it handles similar scenarios to the one proposed here.

The reasons for such a transformation can be diverse. One possible scenario could be the intention to reduce the size of compiled programs. A necessity for this can be found in software development for embedded systems or mobile devices. Due to limited resources available, usually, these fields show critical restrictions on program efficiency, performance, and size. Unnecessary code fragments belonging to certain features can be more easily stripped out if they have been identified priorly. This results in less code, which lastly leads to a smaller binary file and less memory being needed during execution. An additional factor is the reduced amount of necessary control statements. After the transformation all these statements are replaced by control mechanisms executed at the time of compilation. This ultimately decreases the number of necessary evaluation procedures being performed during the execution of the resulting program, which increases the performance.

Independent from the quality of the results of the particular analysis discussed in this thesis, it is necessary to make certain claims upon how transformation must be performed. An automation of this process should also be considered and further investigated to determine the potential of this application case. This is not a trivial task because it needs to be determined, which code artifacts must be annotated for a certain static variant and which need to remain in the source code. A major concern for this is that the resulting source code needs to behave in the same way as it did prior to the modification, with the exception of the extracted feature itself.

By annotating and, thus, disabling the definitions and declarations of the relevant variable selected and identified in earlier steps we need to modify the rest of the code accordingly. All dependencies towards that particular variable need to be addressed in the same way. The annotation process is completed when all dependent code statements were also annotated accordingly. The modified code with the newly created static variability needs to compile and run correctly either way, with and without the feature enabled.

The following examples show how transformation would be performed in the `grep` project when the case sensitity function is to be turned into static variability. All code snippets take place in the file "`main.c`". Example 3.14 shows code fragments that need to be removed entirely and example 3.16 displays more complex code transformations that would be necessary to maintain the same behavior

Listing 3.14: Before removal of case sensitivity.

```
int match_icase;
[...]
while ((opt = get_nondigit_option ...
    switch (opt)
    [...]
    case 'i':
    case 'y':   /* For old-timers . . . */
        match_icase = 1;
        break;
[...]
```

Listing 3.15: After removal of case sensitivity.

```
[...]
while ((opt = get_nondigit_option ...
    switch (opt)
[...]
```

Listing 3.16: Before removal of case sensitivity (`if`).

```
if (match_icase
#ifdef MBS_SUPPORT
      && MB_CUR_MAX == 1
#endif
     )
    {
      int i = lim − beg;

      ibeg = buf = xmalloc(i);
      while (−−i >= 0)
        buf[i] = tolower((unsigned char) beg[i]);
    }
  else
    {
      buf = NULL;
      ibeg = beg;
    }
```

Listing 3.17: After removal of case sensitivity (`if`).

```
#IFDEF FEAT_ICASE
        if(MB_CUR_MAX == 1) {
        int i = lim − beg;

        ibeg = buf = xmalloc(i);
        while (−−i >= 0)
                buf[i] = tolower((unsigned char) beg[i]);
        } else {
        buf = NULL;
        ibeg = beg;
      }
#ELSE
        buf = NULL;
        ibeg = beg;
#ENDIF
```

3.5 Summary

This chapter introduces our analysis approach in detail, which allows the identification of variability in existing source code and an appropriate presentation of the results. First, the general process, which is made up of four phases, is described. Starting with the selection of a particular program behavior, i.e. a configuration option, a controlling variable for this variability is identified manually.

This variable, then, resembles the starting point for an automatic analysis algorithm that restores traceability links for the configuration option to source code fragments. The incremental naive algorithm is based on the concept of slicing and allows the classification of source code statements in three distinguished categories that either implicate a direct affiliation with a feature, no affiliation, or a state of uncertainty. It traverses the AST of a given source code and processes code entities like control statements and functions in a way that control and data flow information can be obtained.

Lastly, these categorization information has to be processed in a certain way, in order to make it valuable to a certain practical application. Two of these are discussed in detail, visual representation, on one side, and code transformation, on the other.

Chapter 4

Implementation

This chapter will cover the implementation of a prototypical tool that allows testing and evaluating our approach to a certain extent on real source code examples. Because of the complexity of the problem and certain time restrictions, we were not able to include all ideas mentioned in the concept explained in chapter 3. One of the main restrictions is the focus on ANSI C source code.

While parsing is handled by the PUMA Framework, which is capable of processing C++, our analysis will only handle basic C statements. Technically C++ can be processed as well, but since control flow and data flow are more complex within the object oriented paradigm, we cannot make any statements on how well the algorithm would perform with it.

4.1 Setting the Scene

Before any analysis can take place, phase one 3.1 and two 3.2 have to be carried out manually. Because the input for the analysis is merely a set of variable names, these have to be selected, identified, and provided in a particular form in advance.

The input for the tool is provided in a file, where each variable name is put onto a line, as displayed in example 4.1. This file needs to be named "featvars" and put into the folder that the tool is executed in.

Listing 4.1: Example of the featvars file for a set of variables in the grep project.

```
match_icase
show_help
color_option
suppress_errors
```

Additionally, usual parsing parameters need to be provided for the PUMA framework. These configurations are needed in certain cases to allow a proper parsing process. Fortunately, most of the needed options can be provided automatically. This is accomplished by using the configuration tool of the AspectC++ project, ac++, with the parameter "`--gen_config`". This configuration file, usually named puma.config, must be provided to the annotator tool as a parameter. A usual execution, therefore, contains a "`--config puma.config`" parameter. To complete the prerequisites, include parameters might be necessary in certain occasions, usually when the project code is scattered throughout several directories. These are either provided in the execution command or added manually to the "`puma.config`" file.

4.2 Using the PUMA Framework

This section covers the role of the PUMA library for the prototypical implementation of the analysis tool. Generally, the tasks covered can be separated into file handling and parsing 4.2.1, which creates the AST of the source code, establishing the semantic database, which holds further information about relations of code entities, and providing a systematic transformation framework to easily alter the AST and reproduce the according code.

4.2.1 Parsing

We use PUMAs project class to hold all necessary source files. This also has the advantage that necessary include files are automatically handled, which prevents unnecessary redundant parsing. A project is linked to a certain root directory, which we provide with the execution command line. A file iterator, which is included in the PUMA framework, further provides us with necessary file handling tools. Combined with file name filters we can track all files that make up a project.

Once the relevant files have been identified, which is done by filtering out all files with a certain file extension, here "`.c`", the parsing process begins. All relevant files are read and a representation in the form of an AST is generated. Because PUMA does have checking mechanics already in place, all files need to be in proper C format. Errors will result in a termination of the parsing process, which might leave the AST in an incomplete state.

Since the framework also takes care of the preprocessing directives, which is useful to further expand the analysis' coverage, only the parts of the source code become part of the AST that belong to the static configuration. Therefore, code fragments enclosed in `#ifdef` statements are not included in the analysis if the particular macro was not defined in this configuration.

The created ASTs are held for reference within the memory in a vector structure. These data structures, named *translation units*, consist of a set of linked source code objects representing either abstract constructs, which define grammatical subdivisions or the tokens themselves, usually at the lower end of the hierarchical structure. Typically, the first entity of a translation unit is the file itself, underneath which the further grammatical constituents appear as children. These objects hold several properties including the type, location, the scope, or related other statements.

Example 4.2 shows a function definition from the `grep` source code. Underneath a view of the resulting AST is located. Each line represents a node, while indents reflects the hierarchy.

Listing 4.2: Source code example (grep).

```c
static void suppressible_error (char const *mesg, int errnum)
{
  if (! suppress_errors)
    error (0, errnum, "%s", mesg);
  errseen = 1;
}
```

```
FctDeclarator
    SimpleName suppressible_error
      Token "suppressible_error"
   ArgDeclList
     Token "("
     ArgDecl
       DeclSpecSeq
         PrimDeclSpec
           Token "char"
         PrimDeclSpec
           Token "const"
       PtrDeclarator
         Token "*"
         SimpleName mesg
           Token "mesg"
     Token ","
     ArgDecl
       DeclSpecSeq
         PrimDeclSpec
           Token "int"
       SimpleName errnum
         Token "errnum"
     Token ")"
  CmpdStmt
    Token "{"
    IfStmt
      Token "if"
      Token "("
      UnaryExpr
        Token "!"
        SimpleName suppress_errors
          Token "suppress_errors"
```

```
     Token ")"
      ExprStmt
        CallExpr
          SimpleName error
            Token "error"
         ExprList
           Token "("
           Integer 0
             Token "0"
           Token ","
           SimpleName errnum
             Token "errnum"
           Token ","
           ImplicitCast
             String "%s" [2]
               Token ""%s""
           Token ","
           SimpleName mesg
             Token "mesg"
           Token ")"
       Token ";"
    ExprStmt
      BinaryExpr
        SimpleName errseen
          Token "errseen"
       Token "="
       Integer 1
         Token "1"
     Token ";"
    Token "}"

[...]
```

Figure 4.1: AST by PUMA

4.2.2 The Semantic Database

An essential part of the library to this implementation is the semantic database that can be created. It is part of a semantic checking functionality that gathers information from the previously obtained AST, puts it into a more accessible data structure, and, lastly, uses this to do basic semantic checks to assure the correctness of the C or C++ source code. This assessment includes type checks, definition checks, or the correct application of `GOTO` and `return`-statements, respectively.

The database contains all existing source code objects, like the files themselves, classes, functions, unions, and even variables. Additionally, it holds more abstract constructs, like scopes, to reconstruct cohesive code blocks corresponding to functions, compound statements, or other similar artifacts. Conveniently, all those objects are organized within easily accessible container objects, which also contain references to the actual objects in the corresponding AST. To increase the usefulness for analysis purposes even more, each of the database objects also contains a specific set of properties. For instance, objects representing functions hold data about return types, parameters, or if it is a static definition. Besides these data structures PUMA also offers a variety of functions to analyze expressions to a certain degree or locate constructs, like variable definitions.

4.3 Putting it All Together

Because we are not only dealing with single files, but rather entire projects, we have to gather certain additional information in advance. PUMA creates translation units and according databases separately for each file. Therefore, we create a combined database with all necessary objects first, which are function definitions and external variables.

4.3.1 Prepare Required Data Structures

The former are needed to be able to follow the control flow of the execution to a certain degree. In many cases the "`main`" function marks the starting point of execution. Its definition can be located in any file and needs to be located first. Additionally, function calls can be made across files, as well. Thus, creating a unified function database is a necessary step to provide the AST parts with function definitions to the algorithm in an efficient way.

Global variables are another important thing to be analyzed in advance. Given the premise that the variable that is interesting to this analysis turns out to be a global one, each code file that uses it has to be analyzed in its entirety right away. The holistic scope of these variables make the consideration of control flow unnecessary, at least for the initial consideration. Practically, we do this by matching the variable names given as input to all external variables that exist in the project files.

4.3.2 Traversing the AST

As mentioned, we have two possible starting scenarios, either any file that uses the feature variable with a global state or a main-function. If the former is the case, the latter becomes unnecessary. We can claim this because in that case the main function is also checked automatically, if it actually uses this global variable. This renders a separate analysis run from the main-function to be redundant and unnecessary, respectively.

Initialization of Algorithm

We created a class that allows us to look up of function names including their definition nodes and initialize the actual recursive algorithm. This class is necessary to set the boundaries of a run. It holds the name of the variable in focus, the starting node of the AST, and several other parameters, i.e. if the variables scope is global or not. This class will always be the starting point for a recursive call upon the investigation of other variables that relevance has been transferred to.

If the variable in focus turns out to have a global scope, a database has been created with all necessary files actually using this variable. This initialization class then starts the analysis on each file. In the case that the feature variable is not used globally, the starting function is located in the function database, and the according AST is visited by the invocation of the recursive analysis algorithm.

Recursive Algorithm

This class uses recursion to visit all nodes, which mostly represent statements on the main level in the hierarchy and individual components of these statements in lower levels. The nodes are visited by a customized class, which was derived from the existing visitor implementation that does the semantic checks and creates the semantic database. This implementation applies the visitor design pattern [Palsberg and Jay, 1998], which defines new operations for several object structures without the necessity of having to alter the object classes.

The recursion is created by a certain function, named `visit()`. It consists of several rules represented by control statements that check the individual type of the particular node passed to it. If necessary, it descends into it by calling a function that corresponds to the individual node type. These functions can then be used to process the individual statements or call the `visit()`-function again with a child node passed as a parameter.

The individual statement functions represent the perfect point to implement our analysis processes because they allow us to treat code entities individually or disregard others that are irrelevant for our investigation.

The relevant nodes we determine and process are the according AST elements representing the control structures that were described in Section 3.3.3. Therefore, appropriate operations handle the statement categorization for all selective and iterative control statements and one operation restarts the analysis algorithm from AST elements that represent function definitions after the according function calls were processed.

4.3.3 Using the Results

To use the results, which are basically a set of categorized code blocks, we decided to apply a visual property to the original source code files. To establish this, we decided to use a prototype of a software product line IDE, called Feature Commander (FC)[1]. It is intended to support programmers in the development process by applying visual highlighting to C source code, which helps the user to find feature-based files or code fragments.

While it is mainly used to highlight feature code, which is usually annotated accordingly, a use for any other type of highlighting is possible. FC uses several files as input to display the source code in a structured manner. The results from our analysis, therefore, are drawn from the data structure after the algorithm finished and used to generate the appropriate files to be displayed in the FC.

The necessary files are:

`models.d`: This file comprises the feature model of all features that exist in the source code. It also describes the relationships of these features. Depen-

[1] `http://wwwiti.cs.uni-magdeburg.de/~feigensp/xenomai/`

dencies and conditions are usually described in the form of rules. Because we do not have these relationships in our consideration, we only generate a minimal file. For FC to function, we, therefore, just add all the entries we need to the project and treat them equally. Each category and variable combination is included, resulting in three possible classifications for each feature, representing a *positive*, *negative*, or uncertain *possible* affiliation with the feature.

`annotations.xml`: This is the standard file for FC that holds the highlighting information for the source code. The XML file contains several necessary elements that define the project, folders, code files, and, lastly, the individual code fragments. The <project>-element simply defines an arbitrary name for the project. It contains a hierarchy of <folder>- and <file>-tags, that hold the according name as an attribute and resemble the physical structure of the source code project on file system level. Within these elements the individual code blocks are contained. They are represented by the <fragment>-element and hold specific beginning and ending line numbers that define coherent lines forming a relevant code block. It also holds a <feature>-element, that defines the name of the feature name and category combination.

Example 4.3 shows a possible file structure taken from the grep-project. The prefix "CONFIG_" is an arbitrary prefix, similar to conventions used with annotations, which are sometimes followed by the category a certain code block was assigned to. The categories we explicitly name are "NOT" resembling the case where a code block must not belong to a certain feature, and "POSSIBLE" referring to all fragments that could not clearly be mapped.

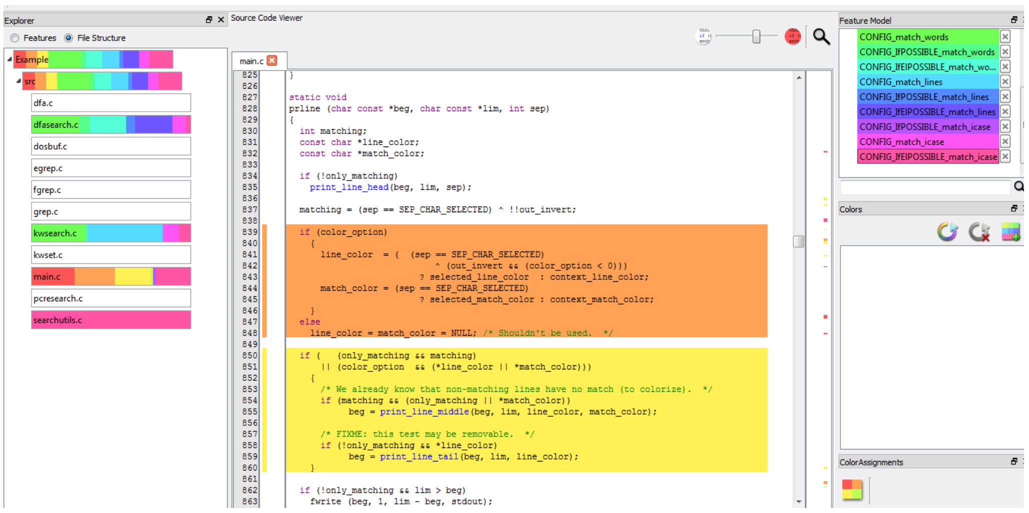
All other statements belong to the actual concern and have no extra tag.

Listing 4.3: Structure of the file annotations.xml for a project named "Grep".

```
<project name="Grep">
        <folder name="src">
        <file name="dfa.c" length="0"></file>
        <file name="dfasearch.c" length="0">
                <fragment startline="71" endline="75">
                        <feature>CONFIG_POSSIBLE_match_icase</feature>
                </fragment>
                <fragment startline="125" endline="126">
                        <feature>CONFIG_match_icase</feature>
                </fragment>
        [...]
        <file name="main.c" length="0">
                <fragment startline="381" endline="382">
                        <feature>CONFIG_NOT_suppress_errors</feature>
                </fragment>
[...]
```

Altogether, the source code files and the two configuration files described above form a presentable project that can be displayed in the FC software. Figure 4.2 shows results we drew from the grep project upon several identified feature variables. The main application screen shows three different views: the *explorer view*, the *source code view*, and the *feature model view*.

The first is located on the left and displays the project structure in the form of a hierarchy of folders and files. These are additionally visualized by rectangles that display the existence of features, either for each individual file, or accumulated for the folders. Because this application was designed for annotated features in source code, certain visualizations are not entirely adequate for our use. In example, the feature bars in the explorer view do not accommodate for lines that have not been assigned to features, thus, leaving the impression features are taking up entire files when they really are not. Nonetheless, this problem can surely be addressed with additional effort.



```c
}

static void
prline (char const *beg, char const *lim, int sep)
{
  int matching;
  const char *line_color;
  const char *match_color;

  if (!only_matching)
    print_line_head(beg, lim, sep);

  matching = (sep == SEP_CHAR_SELECTED) ^ !!out_invert;

  if (color_option)
    {
      line_color  = (  (sep == SEP_CHAR_SELECTED)
                     ^ (out_invert && (color_option < 0)))
                    ? selected_line_color  : context_line_color;
      match_color = (sep == SEP_CHAR_SELECTED)
                    ? selected_match_color : context_match_color;
    }
  else
    line_color = match_color = NULL; /* Shouldn't be used.  */

  if (   (only_matching && matching)
      || (color_option  && (*line_color || *match_color)))
    {
      /* We already know that non-matching lines have no match (to colorize).  */
      if (matching && (only_matching || *match_color))
        beg = print_line_middle(beg, lim, line_color, match_color);

      /* FIXME: this test may be removable.  */
      if (!only_matching && *line_color)
        beg = print_line_tail(beg, lim, line_color);
    }

  if (!only_matching && lim > beg)
    fwrite (beg, 1, lim - beg, stdout);
```

Figure 4.2: Feature Commander showing several identified concerns in the grep-project.

The view located in the center is called the source code view. It displays the actual source code located in the files that were opened from the explorer view. Additionally, it highlights feature affiliated lines of code accordingly. To enhance readability, the transparency of the background colors can be adjusted[Feigenspan et al., 2011]. Bars on both sides of the view indicate where within the file feature code is present. These also indicate when code blocks were assigned to several different features, which is only displayed by one background in the source file to avoid confusion.

Lastly, the *feature model view* is located on the right side of the screen. It shows all existing features in a tree layout. Because we are not creating feature hierarchies, the tree has one level only for our analysis. It also allows for assigning colors to the features individually, which is directly represented in the other views, as well. This leaves the user in charge to more efficiently locate feature related code in the project.

4.4 Summary

With the application of the *PUMA* parsing framework, we were able to implement a naive and conservative analysis algorithm for automatic variability identification. The algorithm incrementally traverses the AST that is created by the parsing functionality. With the help of additional information provided by the semantic database, relevant control statements are processed and categorized. Thus, this tool is able to parse C source code files reliably and, lastly, provides the results in a way that can be displayed in the FC tool for comfortable code navigation.

Chapter 5

Evaluation

In this chapter, we evaluate the results within the context of manual execution of the same task. It is not possible to determine accuracy or completeness for our tool because there is not a single *correct* or *true* answer on what feature code in particular projects is. If certain parts belong to a feature implementation relies on the intentions of particular developers and individual interpretation.

Nonetheless, we can try to look at results we obtained from several projects and compare them, i.e., to a very simple manual approach that uses text matching to look for the same code parts with certain variable names in them. This particularly lets us check the results for the existence of false positives and false negatives, because they mark code points that have either been marked falsely or have been missed by the tool, although, they should have been marked accordingly.

5.1 Test Cases

We tested our tool on three open source projects. Grep [1] is a command-line text and pattern search utility that was originally written for UNIX. The vsftpd [2] project is an File Transfer Protocol (FTP) server implementation that

[1] `http://www.gnu.org/s/grep/`
[2] `https://security.appspot.com/vsftpd.html`

	grep	vsftpd	ngIRCd
Version	2.6.1	2.3.4	0.17.1
Code Files in whole project	238	97	84
LOC of whole project	$\approx$ 39000	$\approx$ 15500	$\approx$ 16500
Files in main source folder	19	97	65
LOC of main source folder	$\approx$ 6100	$\approx$ 15500	$\approx$ 14000
# of Functions in analysis	134	738	537

Table 5.1: Properties of the three open source projects.

was developed with the focus on performance, reliability, and security. The third program is called ngIRCd [3] and represents an Internet Relay Chat (IRC) server application that was built from scratch, unlike many other daemons that are based on existing libraries from the original developers.

Table 5.1 shows general properties of the projects. We distinguish between code that is directly part of the project because the according code pieces are gathered in a folder, and code parts that we believe to be supportive libraries because they are kept at separate locations from the original project code files. Running the tool on both variations created the same results, which further supported our assumption.

Within all the projects we were able to identify almost all occurrences of certain option variables and their covered dependency variables, that we can also find with a standard manual text matching approach. False negatives mostly resulted from static annotations that contradicted with the set up configuration and are processed by the PUMA framework automatically. Example 5.1 shows such a code part with the identified feature variable `case_fold`, that directly represents `match_icase` from the grep code. The first code block is correctly marked, while the second starting on Line 9 is ignored.

[3] `http://ngircd.barton.de/`

Listing 5.1: Example of ignored feature code as a result of static variability that is resolved by PUMA automatically.

```
        if (case_fold && ISALPHA(c))
          {
            zeroset(ccl);
            setbit_case_fold (c, ccl);
            return lasttok = CSET + charclass_index(ccl);
          }
[...]
#ifndef GREP
      if (case_fold && iswalpha(wctok))
        {
          addtok_wc (towupper(wctok));
          addtok (OR);
        }
#endif
```

False positives were also encountered on the same project. Especially the ternary operator seems to be problematic with the use of `PUMAs` semantic database. Example 5.2 shows a code sample that led to the creation of falsely marked code fragments. The variable `case_fold`, here, is used as the decision variable for the if expression leading to a case where either the result of `towlower(wctok)` or the value of `wctok` itself is passed to the function. Unfortunately, the parsing framework identifies `case_fold` itself as the parameter that is passed and replicates this false relationship in the semantic database. This leads to a false investigation of the function `addtok_wc()`, false dependency assumptions on its local variables, and, at the end, falsely identified code pieces.

Listing 5.2: Example of problematic function call resulting in false dependency investigation.

```
addtok_wc (case_fold ? towlower(wctok) : wctok);
```

In all projects the identification of feature code related to boolean configuration variables seemed very accurate. In all tested scenarios the occurred false positives and false negatives could be traced back to similar scenarios as the ones mentioned above. The application on configuration variables that hold integer or string values did also create useful results, comparable to the ones that can be obtained by manual text pattern search. As expected, though, the classification here tends to result in the "possibly belonging to the feature" category in almost all cases. This results from the fact that these variables are more prone to be used in complex conditional statements, which are usually classified in that way by our algorithm. Also, these variables more often are passed to functions or assigned to different variables, which increases the number of possible false positives because these values are actually processed, opposed to the simple boolean values.

Using this tool to analyze entire code projects takes only several seconds. Here, the parsing process consumes the most time usually, as well as the creation of the semantic database. Because this only has to be done once, analyzing several features does not increase the total time dramatically.

The results are directly formatted and written into according files, which can be displayed flawlessly in the FC application. These projects are displayed in their entirety, and according views are created as explained earlier in section 4.2.

This part of the evaluation shows that the prototype is capable of handling entire application projects. The analysis produces quick results that can be viewed instantly, opening several perspectives to inspect the application code with the desired feature code highlighted.

5.2 Case Study

To further determine the value and potential of our approach we decided to conduct a small experiment, which does not provide empirical results but rather represents an interview with two test subjects that have good knowledge in computer science and C programming. These users had the task to identify feature code within two test projects, grep and vsftpd, manually. We did not give them any detailed instructions on what steps to perform, but rather wanted to find out what approach would be used, what detailed dependencies would be considered, and, finally, how efficient the task could be performed if certain configuration options were given.

Because of this, measuring time to compare the approaches does not make sense. The results presented here are qualitative observations that can help us to make according claims rather than measuring and comparing quantitative values.

The first interesting observation we were able to make was that without any direct detailed description all of the test persons picked a similar approach to the first analysis process steps. Especially, the task of finding the configuration variable was solved exactly the same way. Thus, all persons were able to find the relevant feature variable very quickly. This could also be the case because both projects used clearly separated and trivial ways of processing configuration parameters.

In the second phase also similar approaches could be identified. All interviewees did use some sort of text search utilities to look for occurrences of the feature variables. While one person chose to manually search the code files one by one, another used a combination of grep and pipe UNIX console commands. Another option observed was the code highlighting and referencing capabilities of the Eclipse IDE.

In all constellations the test subjects noted the amount of repetitive tasks of searching for an occurrence and marking it according to the nature of the statement. `If`-statements were usually quickly marked as belonging to the feature when the conditions were trivial, or classified as uncertain when the expressions became more complex. Especially, when the analysis came to points where dependencies needed to be traced, i.e. due to variables being passed to functions, a manual inspection became more and more confusing. For instance, keeping track of these dependency trails turned out to be a frustrating practice because non of the used tools provided possibilities to keep record of visited code parts. These more or less repetitive tasks consumed several minutes of time - usually 10 to 15 minutes per feature - and eventually led to very similar approximating results.

While most of the results seemed very similar, two scenarios were identified by two people that our approach cannot process appropriately. Both of them require the evaluation of values that certain variables hold and the consequences these assigned values have on further execution of the source code.

In the first example, 5.2, the variable `syntax_bits` is set to a certain value in dependence of the value in `match_icase`. This value ultimately gets transferred to the variable `syntax_bits_set`, which also has impact on the control flow of this program. Thus, the control statement on Line 5 is directly dependent on `match_icase` and should be marked accordingly.

```
1  if (match_icase)
2      syntax_bits |= RE_ICASE;
3    re_set_syntax (syntax_bits);
4    [...]
5  if (! syntax_bits)
6      dfaerror(_("no syntax specified"));
```

A different scenario is shown in example 5.2. Here the expression belonging to the `if`-statement in Line 1 directly influences the statement block underneath. The condition obviously resolves to `true` whenever `match_icase` holds a value equivalent to `false`. This means that the `return`-statement in the next Line is executed and the following code block is never reached. Vice versa, whenever `match_icase` holds the value `true`, the `return`-statement is mostly not executed rendering the following code block to be necessary. Even though the condition holds a more complex expression, a direct affiliation of the code block starting on Line 4 and `match_icase` exists. Therefore, a more correct result should highlight that code block accordingly.

```
1  if (MB_CUR_MAX == 1 || !match_icase)
2      return execute(buf, size, match_size, start_ptr);
3
4    for (line_next = buf; line_next < buf + size; )
5      {
6        const char *line_buf = line_next;
7        const char *line_end = memchr (line_buf, eolbyte, (buf + size) - line_buf);
8        if (line_end == NULL)
9          line_next = line_end = buf + size;
10       else
11         line_next = line_end + 1;
12
13       if (start_ptr && start_ptr >= line_end)
14         continue;
15
16       result = execute (line_buf, line_next - line_buf, match_size, start_ptr);
17       if (result != (size_t) -1)
18         return (line_buf - buf) + result;
19     }
20
21   return (size_t) -1;
```

This interview shows clearly that our analysis process has similarities with the approach developers would take. Additionally, we can claim that the prototype can create valid results in less time than what an entire manual execution can accomplish. Because there are scenarios that are prone to create errors, a subsequent investigation of the results is necessary. This supports the current intention of using the tool as a support for developers in maintenance and code comprehension tasks. This case study also showed where the limitations of this simple prototype algorithm are. The problematic scenarios that were identified in the context of this interview also point to the assumption that the application of partial evaluation and abstract interpretation techniques could improve the accuracy of the approach.

Chapter 6

Problems and Future Work

Slicing and the related static analysis concepts span a wide ranging and complex research field. The restrictions that come with undecidable properties of source code analysis (see 2.2.1), of course, also apply to the identification of features, which relies on it. As a result, it was not, and probably never will be, possible to achieve completely accurate analysis results from the prototypical implementation.

Thus, our focus was to create a prototype as a proof of concept. This section, therefore, gives a brief overview of the problems that arose during the implementation. On one hand, these are issues related to the approximation and, on the other hand, technical limitations in the PUMA framework.

6.1 Problems Concerning Approximations

The procedural programming language C, which is significantly less complex than its object-oriented extension C++, already holds challenges for analysis tasks. Therefore, we had to make restrictive decisions for the initial implementation to limit the amount of development, which resulted in even less accurate approximations at the end. Nonetheless, some of these restrictions are not entirely inevitable. The implementation of more complex static analysis approaches or the usage of additional information can increase accuracy to a certain degree.

6.1.1 Dependency Resulting from Variable Manipulation

A major problem in determining feature-related source code fragments is the determination of range that certain dependencies may have. While some dependencies are clear, others are very subtle and rely on extensive evaluation of scopes, possible values held by variables at particular points, and special code structures. During the implementation phase we noticed that with our simple algorithm we are not able to cope with a certain type of dependency.

One case is created when data items are modified within the source code fragments of a feature which might have an impact on control flow during further execution. A variable modification within the feature code is directly dependent on the fact if the feature is enabled and, thus, the according code is executed. Following control statements that rely on these variables can only be mapped to features by investigating the possible values that this variable can hold at that particular point.

Example 6.1 shows such a scenario. In a subtle way the control was passed from `feat_A` to the data item `datatype`. A programmer can notice the direct relationship between `feat_A` and and the compound statement on Line 7, though, he cannot directly make a decision about the categorization of the `compound_statement`. But, our prototype cannot cope with this dependency because it does not consider the values that any of the data items can hold at that time. These are necessary to make any assumptions about the affiliation of this code part with `feature A`. A programmer also needs to check this dependency by determining possible values for this variable manually. In the case that `KEY_VALUE` can only be assigned to `datatype` within this displayed part of feature code, the `compound_statement` can directly be mapped to `feature A`.

Listing 6.1: Example of dependency on data values.

```
if(feat_A) {
        datatype=KEY_VALUE;
        [...]
}

if(datatype==KEY_VALUE) {
        [compound_statement]
}
```

Even with more precise dependency analyses one can only determine that a relationship exists, not the nature of the dependency. A simple method to at least address this relationship would be to gather all variables that are modified within feature code and have them analyzed in more detail in a subsequent step.

To cope with this kind of problem in a more beneficial way, practical solutions need to incorporate some kind of value analysis. Existing methods to do this can be found in the field of partial evaluation or symbolic execution [King, 1976] (see 2.2.2). With a theoretical execution of certain code parts more precise claims can be made, which result in more accurate categorizations.

We found potential solutions like the Frama-C[1] analysis framework to be a valuable tools to look at in the future. Multiple extensions allow to determine possible values that variables can hold at specific points in the code. Therefore, this information can be used to compare potential values in the variable `datatype` within the first and the second compound statement, which, when compared, show the dependency clearly.

[1] `http://frama-c.com/`

Because Frama-C was written in the programming language OCaml[2], it is not easy to integrate it into our analysis prototype. While it showed accurate and, therefore, interesting results on simple examples, we were not able to use it for bigger software projects, due to parsing errors and unresolved file dependencies. Nonetheless, we found that partial evaluations are possible to a certain extent, which should be looked at in the future to possibly further enhance our prototype.

6.1.2 Increasing Accuracy of Control Flow Information

As mentioned in chapter 3, we are not able to process the more complex control flow mechanisms, like function pointers or `GOTO`-statements. Even though their use within the code is quite simple the term complex, here, refers to the challenging task of analyzing possible effects and dependencies.

Function pointers represent a useful mechanism to implement dynamic variability and, thus, should be further investigated. Pointer analysis approaches [Ghiya and Hendren, 1998, Wilson and Lam, 1995] can be used to address the problem that this concept might have on relevant code parts in our analysis algorithm. Because the effects and consequences the application of pointer can have on a program are very diverse, an independent exploration of these mechanisms in the context of our approach could prove to be valuable. We did find some application of this control mechanism in the investigated projects, but they were not related to any of the configuration variables that we looked at. A future step should first evaluate how common this practice is for the implementation of dynamic variability. If it proves to be beneficial, a suitable pointer analysis approach needs to be added to the analysis algorithm.

[2] `http://caml.inria.fr/ocaml/index.en.html`

GOTO-statements create a big challenge in terms of determining the scope of effect and the exact conditions under which certain code parts are reached. This problem is closely related to an issue that arises from the use of multiple return-statements. An observed scenario is described in section 5.2 (example 5.2). Similar to the problem in section 6.1.1 possible solutions require the application of partial evaluation to determine the exact conditions.

Further, static analysis tools, like GNU cflow, already provide more precise control flow information on C programs than what we achieve in our prototype. The inclusion of such analysis results from external tools could increase the accuracy of our analysis. Future work should investigate and evaluate how our approach could benefit from better control flow analysis methods, either provided externally by other tools or internally by improving the current algorithm.

6.2 Parsing Problems

The PUMA framework allows us to parse ANSI C code and provides several built-in mechanisms to deal with semantics and preprocessor directives. During our tests we encountered several problems that we could trace back to inconsistencies in the parsing libraries implementation.

With this framework we are limited to a single code configuration that we can analyze. PUMA does preprocess the code automatically, which also means that certain compile-time configurations cannot be included. Specifically, we found that many projects use a certain header file (mostly named "config.h") that holds all necessary preprocessor definitions. As a result our parser also processes this configuration, which means only the current particular static variant defined by the according preprossor directives is parsed. Therefore, we are not able to analyze the entire source code structure including all possible static variants unless a configuration exists that comprises all of them at once.

Example 6.2 shows such a structure. In a case where both annotated structures cannot be enabled at the same time, i.e. because they implement two alternative static variants. Within a configuration file either `define1` or `define2` is defined, and, therefore, the corresponding code blocks will be included in the parsing process. Code blocks annotated with the other define name will be left out and code blocks that potentially belong to a feature implementation cannot be identified.

Listing 6.2: Compile-time variability that is not parsable by the prototype.

```
#ifdef define1
        if(feat_A) {...}
#endif

#ifdef define2
        if(feat_A) {...}
#endif
```

Within our evaluation we also noticed issues that are related to the use of the ternary operator within condition expressions. In the source code example 5.1 we observed unexpected behavior from the parsing library concerning the creation of the semantic database. To address this problem the developers of the framework should be contacted or an individual fix needs to be developed.

Lastly, because the PUMA library relies on a configuration file, the quality of the parsing process strongly depends on how suited a configuration is for a particular source code. We found several C projects that used slightly different dialects, which was either not compatible with the PUMA library or required a specific configuration flag that was not automatically determined by ag++. Thus, parsing of these projects resulted in errors, which generates incomplete ASTs for the analysis process.

Such a project was the Berkely DB[3], which partially uses the function declaration syntax of original C, which is different to the one in ANSI C. Example 6.3 shows the difference in syntax. With appropriate configuration adjustments the PUMA framework would be able to cope with this syntax, yet, we were not able to find it due to the very limited documentation that is available for PUMA at this point in time.

Listing 6.3: Difference in function definition syntax between original C and ANSI C.

```
/* original C */
int
functionName( param1, param2 ) ;
char *param1;
int param2;
{ ... }

/* ANSI C */
int
functionName( char *param1, int param2 );
{ ... }
```

To improve the compatibility of our tool one of the first future steps should include a more precise investigation of the configuration options for the PUMA library. This ensures that more projects can be reliably parsed and analyzed in the future.

[3] http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html

Chapter 7

Related Work

This thesis presents an analysis process to automatically identify selected features in program code. To accomplish this, we combine several steps and analysis approaches. This chapter represents a brief overview of multiple related concepts and research fields.

Program Configuration Analysis. Because we focus on configurable features within software, the effects those have on program behavior is also an interesting property to consider. *Symbolic Evaluation* allows the analysis of these relationships and helps to reduce the amount of configuration scenarios that need to be considered [Reisner et al., 2010].

To further automate process steps, approaches for automatic extraction of configuration options from program code, as proposed by Rabkin and Katz [2011], seem valuable. Even though this concept is not directly tailored to C it does pose great insight into this area of research and might provide the basis for further investigations.

Mining Approaches. Several different approaches and techniques for investigating legacy code exist, but they operate on different levels of granularity, on different inputs, generate different results, or follow slightly different goals and intentions. These are used with several intentions in mind, like code reuse, maintenance, or comprehension. With all differences and similarities, it is not easy to categorize all of these approaches clearly. Often a major difference can be identified in what is investigated in the according analysis.

One concept that shows several similarities to the analysis introduced in this thesis is called *Variability Mining* [Kästner et al., 2011]. It describes four steps that represent a general process to identify feature related program fragments in legacy code. In contrast to our approach, it uses domain knowledge to enhance further investigations.

Feature Mining Dreiling [2010] is another similar concept that specifically uses a feature model and several extraction techniques to migrate legacy code towards software product lines. In his thesis he gives an introduction to the general concept of feature mining. Opposed to other concepts, i.e., it defines strict properties, like fine granularity and completeness.

Component Mining [Spinellis and Raptis, 2000] focuses on the identification of reusable components, in legacy programs, which is similar to the goals of *Asset Mining* [Eisenbarth and Simon, 2001], where code bases are only partially examined through the application of an opportunistic ad-hoc decision algorithm. Another approach operating in a different dimension is *Aspect Mining* [Godil and Jacobsen, 2005, Störzer et al., 2006], which aims toward identifying source code artifacts belonging to cross cutting concerns (see 2.1.2).

With the exception of Asset Mining, these approaches consider entire programs as input and rely on the code structures and patterns that are investigated, which is opposed to the focus on specific features and the incremental code investigation in the concept described in this thesis. The nature of the dynamic parameters allow a logical line of actions to conduct this analysis corresponding to the controlling properties of these parameters.

Refactoring. Refactoring describes the process of restructuring source code in a way that does not alter the external behavior of the resulting program [Fowler and Beck, 1999]. This field of research provides developers with tools and algorithms for restructuring source code reliably. These transformations were investigated on several levels, i.e. architectural or within annotations [Tansey and Tilevich, 2008].

Specifically within the concept of features, efforts have been made to construct reliable methods for code refactoring. This specific approach is called *Feature Oriented Refactoring* [Liu et al., 2006]. Further, the automation of this process was evaluated in other related work [Kästner et al., 2007].

These theories provide further insight for the transformation phase that was discussed as a processing step of the results. With the distant goal of turning dynamic variability into static variability, suitable refactoring approaches will become necessary and, therefore, need to be investigated.

Chapter 8

Conclusion

In this thesis we presented an analysis process that allows the identification of feature implementations in C source code. This is done by recognizing existing dynamic variability that implements run-time configurations in program source code. We discussed the application of different analysis approaches and found that a slicing algorithm is a suitable method to realize this task. Then, we designed and implemented a tool that uses these theoretical constructs to identify source code fragments on the basis of given variable names, which were selected in advance.

Our approach identified four major steps. Beginning with manual selection and code mapping, we lead towards the actual analysis algorithm that we build upon a naive incremental dependency check, which mimics maintenance and code comprehension tasks that developers would apply manually. The focus on configuration elements and according features reduce the complexity of analysis, immensely. Additionally, we investigated possible utilization scenarios of the results. While code transformation is discussed to a certain extent, the visual representation of the results is also integrated into the implementation.

With an experimental evaluation we were able to compare the results that we obtained from our tool to actual results from developers that manually tried to identify feature code. Not only were we able to see that the approaches seemed very similar, we also found that the naive approach in the prototypical implementation can lead to similar results in less time. Although the developers were able to identify more accurate code dependency types, these could also be investigated after the application of our tool.

Generally, we were able to show that even without the implementation of complex and resource intensive static analysis methods it is possible to support development tasks and increase efficiency and code comprehension. Yet, more accurate results can be achieved with the refinement of the analysis algorithm and extend it with certain dependency identification mechanisms. Especially the automation of repetitive investigative tasks can support developers and increase efficiency, even though the tools only use naive and lass accurate algorithms that produce results that represent less accurate approximations.

Extending and, therefore, improving the analysis process by using more complex static analysis methods, like partial evaluation, marks the direction of future work (see chapter 6). When the approach is made more reliable and accurate the focus can be shifted towards how the results can be used so developers can benefit from this work.

The prototype is already capable of supporting developers in code maintenance tasks that require the identification of code dependencies. In the future more valuable tools could be created that support developers in code navigation and comprehension in a visual way. Highlighting related code, a specific selected variability, or code that seems to implement certain features dynamically would mark the next step that should be taken.

With more reliable and more capable analysis approaches, it could even become possible to create code transformation tools that further reduce the necessity of developer interaction. More accurate analysis results make less manual investigation steps necessary that developers have to undertake while extracting reusable parts from legacy code. Identifying dynamic variability automatically a transforming it in desired ways would create another benefit.

Bibliography

F. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

L. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

S. Apel. *The role of features and aspects in software development*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, Universitätsbibliothek, 2007.

G. Arango and R. Prieto-Diaz. Domain analysis concepts and research directions. *Domain analysis and software systems modeling*, pages 9–26, 1991.

M. Babar, L. Chen, and F. Shull. Managing variability in software product lines. *Software, IEEE*, 27(3):89–91, 2010.

L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., 2003.

F. Buschmann, K. Henney, and D. Schmidt. *Pattern-oriented software architecture: On patterns and pattern languages*, volume 5. John Wiley & Sons Inc, 2007.

A. Businger. *PORTAL language description*, volume 198. Springer, 1988.

K. Chen and V. Rajlich. Case study of feature location using dependence graph. *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 241–247, 2000.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.

Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 238–252, 1977.

P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 170–181, 1995.

K. Czarnecki. *Generative programming: Principles and techniques of software engineering based on automated configuration and fragment-based component models.* PhD thesis, Computer Science Department, Technical University of Ilmenau, October 1998.

E. Dijkstra. *A discipline of programming.* Prentice-Hall series in automatic computation. Prentice-Hall, 1976.

A. Dreiling. Feature Mining: Semiautomatische Transition von (Alt-)Systemen zu Software-Produktlinien. Diploma thesis, University of Magdeburg, 2010.

T. Eisenbarth and D. Simon. Guiding feature asset mining for software product line development. *Proceedings of the International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing, Erfurt, Germany, Fraunhofer IESE*, pages 1–4, 2001.

T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, pages 210–224, 2003.

L. Etzkorn and C. Davis. Automatically identifying reusable oo legacy code. *Computer*, 30(10):66–71, 1997.

J. Favre. Preprocessors from an abstract point of view. *wcre*, page 287, 1996.

J. Feigenspan, M. Schulze, M. Papendieck, C. Kästner, R. Dachselt, V. Köppen, and M. Frisch. Using background colors to support program comprehension in software product lines. *Proc. Int'l Conf. Evaluation and Assessment in Software Engineering (EASE)*, pages 66–75, 2011.

M. Fowler and K. Beck. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 133. ACM, 1998.

I. Godil and H. Jacobsen. Horizontal decomposition of prevayler. *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 83–100, 2005.

O. Gotel and C. Finkelstein. An analysis of the requirements traceability problem. *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 94–101, 1994.

M. Harman and R. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

N. Heintze and J. Jaffar. A decision procedure for a class of set constraints. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 42–51. IEEE, 1990.

F. Henglein. Simple closure analysis. *DIKU Semantics Report D-193, DIKU, University of Copenhagen, Universitetsparken*, 1, 1992.

M. Hermenegildo. An integration of partial evaluation in a generic abstract interpretation framework. *University of Aarhus*, Jan 1999.

W. Hürsch and C. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.

I. Jacobson, M. Griss, and P. Jonsson. *Software reuse: architecture, process and organization for business success.* Addison-Wesley Professional, 1997.

N. Jones, C. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation.* Peter Sestoft, 1993.

C. Kaestner. CIDE: Virtual Separation of Concerns, Mar. 2010. URL `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide`.

K. Kang. Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document, 1990.

C. Kästner. *Virtual Separation of Concerns*. PhD thesis, Universität Magdeburg, 2010.

C. Kästner, M. Kuhlemann, and D. Batory. Automating feature-oriented refactoring of legacy applications. *Suites. The TRex TTCN-3 Refactoring and Metrics Tool 3*, page 62, 2007.

C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. *Proceedings of the 30th international conference on Software engineering*, pages 311–320, 2008.

C. Kästner, A. Dreiling, and K. Ostermann. Variability mining with leadt. Technical report, Philipps University Marburg, September 2011.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. *ECOOP'97—Object-Oriented Programming*, pages 220–242, 1997.

J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. *Proceedings of the 28th international conference on Software engineering*, pages 112–121, 2006.

J. Midtgaard. *Control-flow analysis of functional programs*. BRICS, Department of Computer Science, University of Aarhus, 2007.

C. Mossin. Exact flow analysis. *Static Analysis*, pages 250–264, 1997.

F. Nielson, H. Nielson, and C. Hankin. *Principles of program analysis.* Springer, 1999.

P. Northrop. *Software Product Lines: Practices And Patterns(Series - Sei Series In Software Engineering).* The SEI series in software engineering. Egully.com, 2007.

K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 19(5):177–184, 1984.

J. Palsberg and C. Jay. The essence of the visitor pattern. *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*, pages 9–15, 1998.

D. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, pages 1–9, 1976.

K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques.* Springer-Verlag New York Inc, 2005.

A. Rabkin and R. Katz. Static extraction of program configuration options. *Proceeding of the 33rd international conference on Software engineering*, pages 131–140, 2011.

E. Reisner, C. Song, K. Ma, J. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 445–454, 2010.

D. Spinellis and K. Raptis. Component mining: A process and its pattern language. *Information and Software Technology*, 42(9):609–617, 2000.

M. Störzer, U. Eibauer, and S. Schoeffmann. Aspect mining for aspect refactoring: An experience report. *Towards Evaluation of Aspect Mining—TEAM 2006—*, page 17, 2006.

M. Svahnberg and J. Bosch. Issues concerning variability in software product lines. *Software Architectures for Product Families*, pages 146–157, 2000.

W. Tansey and E. Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. *ACM SIGPLAN Notices*, 43(10):295–312, 2008.

P. Tarr, H. Ossher, W. Harrison, and S. S. Jr. N degrees of separation: Multi-dimensional separation of concerns. *Proceedings of the 21st international conference on Software engineering*, pages 107–119, 1999.

F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

M. Urban. Ein Codeanalysesystem für die Familie der C-basierten Sprachen. Diploma thesis, Otto-von-Guericke-Universität Magdeburg, Germany, December 2002.

M. Urban, D. Lohmann, and O. Spinczyk. The aspect-oriented design of the PUMA C/C++ parser framework. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 217–221. ACM, 2010.

F. Van Der Linden, K. Schmid, and E. Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Springer-Verlag New York Inc, 2007.

M. Weiser. Program slicing. *Proceedings of the 5th international conference on Software engineering*, pages 439–449, 1981.

J. G. Wijnstra. Supporting diversity with component frameworks as architectural elements. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 51–60, New York, NY, USA, 2000. ACM.

R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI 1995 Proceedings of the ACM SIGPLAN 1995 conference* *on Programming language design and implementation*, volume 30. ACM, 1995.

Abschliessende Erklärung

Ich versichere hiermit, dass ich die vorliegende Diplomarbeit selbständig, ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Magdeburg, den 08. Dezember 2011